2016

Crawling and Analyzing Repository in GitHub

Zhongpei Zhang
University of Windsor

Crawling and Analyzing Repository in GitHub

By

Zhongpei Zhang

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science
at the University of Windsor

Windsor, Ontario, Canada

2016

Crawling and Analyzing Repository in GitHub

by

Zhongpei Zhang

APPROVED BY:

H. Wu
Department of Electrical and Computer Engineering

X. Yuan
School of Computer Science

J. Lu, Advisor
School of Computer Science

September 16, 2016

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyones copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

GitHub has become one of the most popular online software developing website. I have crawled the most popular software repositories (own over 500 star number) in GitHub, along with their contributors and stargazers. In total, we have crawled 10,665 repositories, 176,256 contributors, and 1,170,449 stargazers. One of the most important missions of analyzing is detecting communities from the network. While the heterogeneous Github network includes three objects, user, repository and programming languages and two kinds of relation between user and repository, i.e., star and contribute. Mining heterogeneous information network is a fresh and promising research field in data mining. A lot of algorithms has been proposed for heterogeneous network clustering. However, most of these methods directly cluster the heterogeneous networks. This thesis aims to transform the heterogeneous network to the homogeneous network using different schemes and then cluster the new network. We studied three weighting schemes, including dot product, Jaccard similarity and cosine similarity between the vector representations of objects. Then I cluster the homogeneous network by using modularity maximization optimization algorithms, in particular, greedy modularity maximization optimization algorithm and spectral modularity maximization optimization algorithm. The performance of clustering is evaluated using F-measure and rand index based on the programming language the software repository used. To compare the interaction between the weighting schemes and clustering algorithms, we applied out methods on GitHub dataset. Then we transformed the whole network to repository-repository and furthermore transformed it to the language-repository network. Based on this network, we discovered the relation between languages. Among 94 programming languages used by the top 10,000 projects, we studied their relations using several clustering methods. Overall, we find that languages fall into five communities, i.e., web and scripting languages (JavaScrip, HTML, etc.), system programming languages (C, C++, etc.), OS X and IOS programming languages (Objective-C, Swift, etc.), numerical and statistical languages (Matlab, FORTRAN, Julia and R), and functional programming (Lisp, Scheme, etc.).

ACKNOWLEDGEMENTS

I would like to present my gratitude to my supervisor Dr. Jianguo Lu for his valuable assistance and support during the past two years.

I also would like to express my appreciation to Dr. Xiaobu Yuan, and Dr. Huapeng Wu. Thank you all for your valuable comments and suggestions to this thesis.

Finally, I want to thanks to my parents, my friends who give me consistent help over the past two years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

Introduction

Nowadays, GitHub[1], an open source website for software repositories, has become increasingly popular and attracts a great number of software developers around the world. It provides the services of code hosting, as some platforms have done before, such as SourceForge, BitBucket, and Aseembla. Different from these websites, it more focuses on the social features. Actually, GitHub could be seen as a storage of codes but also as a free and easy-to-use on-line tool for collaborative software development. Besides, it also offers a lot of functions to support the community of developers. The network of the users of GitHub can be regarded as a small-scale social network. In this network, the developer can follower those who attracts their attention. The GitHub provide a function called event, which may send an intermediate notification of the latest information about what happen to their following developers to the developer. Moreover, in GitHub, any user can create their own code repository. Every repository can be developed by more than one developer. The owner of the repository can add other collaborators and invite them to complete the repository together. But they are not the only one who can change the code of the repository. In fact, every developer who wants to join the development of the repository can make the contribution to it by forking. This action copies all the files of the repository to those who fork it, which allows the developers work independently without changing the original code. When the developers complete a new task, like fixing a bug, they can send pull requests to the owner of the original repository. Then the owner can review the changes that they have made and decide whether or not apply these changes in the original repository. Once the changes are accepted, the author becomes one collaborator of the original

repository. The GitHub not only be used to develop software but also can be seen as a resource to search for high-quality software. The users can find out the repository which they are interested in and star and star it. Then when the software updates, they may get the latest information about it. The GitHub also provides download, clone in desktop and some other functions. For April 2016, GitHub claims that it has over 14 million users and over 35 million repositories, which makes it becoming the largest host of source code in the world. Therefore, mining data from GitHub and analyze these information has become an important study subject.

An important task in network data mining is clustering and it is also a classical object-related mission in network mining. Object clustering on networks, which is also known as community detection[9] or group detection [10], aims to distribute objects in a dataset to either separate or overlapping clusters[14] based on their relationship and similarity. Each cluster could also be call community[26, 11] or group [10]. (Note that the terms communities, cluster and group are same in this work.) Clustering has been widely studies for several years in various areas, such as machine learning, pattern recognition, and data mining. Unlike traditional attribute-based clustering techniques[17, 19, 13, 40], clustering based on the network structure also consider the links between objects. So clustering on the network is also called like-based clustering[29] or relational clustering[39]. For homogeneous network clustering, we can directly apply the traditional clustering methods. While applying the traditional methods on the heterogeneous network usually leads to terrible results and this problem has drawn a lot of attention in recent years. One solution for this problems is directly clustering the heterogeneous network and simultaneously cluster objects of each type, but this method is not available for a large network. Another one is heterogeneous-transformed homogeneous clustering [16], which means first transforming the heterogeneous network to homogeneous network and then clustering the homogeneous one. There has been a lot of researches about the directly clustering methods, but less attention has been paid on the heterogeneous-transformed homogeneous clustering.

In my thesis, I cluster the programming languages and investigate the relation

between programming languages. Categorization of programming languages is an important approach to understanding languages. Languages can be clustered according to a variety of criteria, such as by programming paradigms (e.g., functional programming vs. OO programming), by genealogy (e.g., C is the ancestor of C++), by applications (e.g., numerical analysis vs. logic inferences), or by syntax similarities (e.g., Java vs. C#). Traditionally, categorizations are done based on anecdotal accounts, which are hard to verify and quantify.

Thanks to the availability of open source projects such as GitHub, relations between languages can be derived from user interactions. GitHub lists the language for each project and the users who favour the project. Assuming that projects favoured by the same person are related in some way, the relationship between projects can be derived based on the common users they share. Based on the relations between projects, the language relation can be simply the sum of relations of repositories that use the language.

Quantifying the relation is not a trivial task. Existing similarity metrics can not be applied directly. For instance, Jaccard similarity tends to biased towards popular repositories/languages–the more popular the repositories/languages are, the higher similarity they have. To solve this problem, we represent each repository as a vector of users, and the similarity is the cosine of the vectors. Another problem is caused by very active users who give stars profusely to many projects. Their weight should be reduced. Following the IDF(Inverse Document Frequency) heuristic in information retrieval, where popular words are discredited inversely proportional to their document frequencies, we also suggest reducing a user's weight inversely proportional to its stars he gives.

Considering these two observations, we propose to use doc product-idf as the similarity function between two vectors of projects. To verify such weighting scheme, we evaluate it again others, including dot product, cosine, Jaccard Similarity, and cosine-idf. We cluster projects using modularity maximization algorithm. The ground truth is labeled by languages: two projects belong to the same cluster is they are labeled by the same language. Evaluating the result by F-measure and rand index,

the experiment indicates that dot product-idf is the best weighting schemes.

We have applied the above method on the dataset we have crawled. For the data collection, I have crawled the whole list of GitHub users in May 2015. Then based on the user information, I collected the follower information and those repositories with at least 500 stars. I also collect the contributor and stargazers of these repositories and create two networks, repository-contributor network, and repository-stargazer network. As the repository-stargazer network provides more information than the repository-contributor network, we used dot product-idf as the weighting scheme to reveal the relationship between communities of the repository-stargazer network. We separately detected 7 communities for greedy modularity algorithm and 4 communities for spectral modularity algorithm. From the result of both clustering algorithm, we found that there are some programming languages are usually belong to the same community, such web development programming languages (JavaScript, HTML, CSS and PHP), system programming languages (Python, C, C++ and Go) and programming languages for OS X and iOS system (Objective-C and Swift).

We also investigated the relation between programming languages on three different clustering methods. These methods separately based on the high dimensional language-repository matrix, 2 dimensional language-repository matrix, and language-user matrix. From the experiment, it is difficult to decide which methods better performs the relation between programming languages, but we found some common phenomenon. Some pairs of programming languages are usually close, such as HTML&CSS, C&C++ and Matlab&FORTRAN. The results can also have direct commercial applications. For instance, it can be applied in recommending programming related products, such as books, courses, jobs etc. When a programmer buys a Fortran book, she may be also interested in Matlab books. In addition, we find that languages fall into five communities, i.e., web and scripting languages (JavaScrip, HTML, CSS, etc.), system programming languages (C, C++, Python, etc.), OS X and IOS programming languages (Objective-C, Swift, Objective-C++, etc.), numerical and statistical languages (Matlab, FORTRAN, Julia and R), and functional programming (List, Scheme, Racket, Haskell, etc.).

Main contributions of the thesis can be summarized below:

- We crawled a large network that contains millions of users and 10,000 top repositories. It is extremely time consuming to identify the top stared repositories.
- When transforming heterogeneous network to homogeneous network, we propose to use the IDF heuristics. The weighting scheme improves the clustering result significantly.
- We cluster programming languages using a variety of methods, including based on repositories, users and the reduced dimensionality of repositories. And our study surface some deeper knowledge between programming languages, such as FORTRAN and Matlab, Groovy and IDL are usually used in one repository.

For software companies, the knowledge would help them to select the best combination of languages to support. For software developers, they can logically arrange the learn of programming languages. The results can also have direct commercial applications. For instance, it can be applied in recommending programming related products, such as books, courses, jobs etc. When a programmer buys a Fortran book, she may be also interested in Matlab books.

The remainder of this thesis is structured as follows: In chapter II, we introduce the previous works on heterogeneous network clustering and recent research on GitHub dataset. In chapter III, we detailedly explain the process of crawling data from GitHub and describe the detail of our dataset. In chapter IV, we present the three weighting schemes we used to transform the heterogeneous network to homogeneous network. Then in chapter V, we introduce the two clustering methods based on the modularity maximization optimization strategy and evaluation measures. And we apply the methods on the subdatasets extracted from the dataset we collected. In chapter VI, we applied the suitable weighting schemes on the whole network to find out the relation between repositories and then in chapter VII we analyze the relation between programming languages. Finally, we summarize our work, give out the conclusions and describe the future work in chapter VIII.

CHAPTER II

Review of the Literature

Association between programming languages has been studied using data from open sources such as SourceForge [8], GitHub [31] and StackOverflow [37]. One approach is to use language co-occurrence in projects. Deloray et al. found that the closest pairs are (C, Perl), (C, C++), (Javascript, PhP). Karus and Gall [18] found associations such as (Java, XML). (C, make), (JavaScript, CSS) [18]. Language co-occurrence can not reveal the similarity between languages. Similar languages may not co-occur in a project. Matlab and R are similar. In a project, either one of them is used, not both at the same time. The other approach is to use user interactions. Vasilescu et al. [37] use StackOverflow data to study the similarities between language pairs based on user's language tags [37]. Sanatinia and Noubir [31] use user commit data in GitHub to study language relations. They focus on the top 10 languages and use k-means to cluster the programming languages. In this chapter, we will detail these related work.

1 Programming Language Trends in Open Source Development: An Evaluation Using Data from All Production Phase SourceForge Projects

Delorey et al. [8] analyzed projects from SourceForge to realize the trends in programming language.

Dataset Delorey et al. [8] collected the CVS repositories Open Source projects hosted on Source Forge and data from SourceForge Research Archive(SFRA). They

have totally crawled 9,997 projects and 23,83 authors. All these information are written in more than 7.5 million individual files and these files are distinctly changed over 25 million times.

Result and Conclusion For the popularity of programming language, Delorey et al. [8] found that web development languages becomes more popular, but the traditional desktop development languages are decreasing in popularity. Besides, although there is just a little of people like scripting languages, the number of these people remain consistent. The author also studied the use of multiple programming languages for individual projects and individual users. They claimed that only nearly tenth using three languages, less than a quarter using two languages and over two thirds using one language. At last, they believed that each year, the three most common languages profile were single language profiles. While for profiles written by two language, the most popular ones are web development profiles.

2 On GitHub's Programming Languages

Sanatinia and Noubir [31] studied the popularity of programming languages and existence of trends in the relations between users, repositories, and programming languages on Github.

DatasetSanatinia and Noubir [31] designed and implement a resilient distributed data collection system, which inlaces 200 collection advantage points to collect data from Github to avoid the limitation of Github API, that each hour can only sending 5000 requests. In total, the authors collected information of nearly 10 million (9,993,767) users and less than 17 million (16,812,452) repositories. For the user information, over 95% are user account and the rest are organization account. For the repositories, they are designed by over 200 different programming languages, which is denoted by Github.

Method To identify the relationship between top 10 popular programming languages, Sanatinia and Noubir [31] separately calculate the correlation between programming languages by the commits made by users and the repositories built by

each users. Then utilizing the correlation as input, they clustered top 10 programming languages on Github by k-means clustering[3]. Then the authors selected top 100 repositories of each top 10 programming languages and they created a directed graph to represent the relationship between the programming languages. In the graph, each vertex represents a programming language and the edge between each two vertices means whether these two programming languages are simultaneously used in one repository. Then they ran PageRank algorithm[28] on the graph to rank the programming language.

Result and Conclusion From the collecting data, Sanatinia and Noubir [31] claimed that JavaScript is the most popular programming language in Github, and it has more than 7 million stars, which is over twice that the second one. Through the experiment of top 10 programming languages, Sanatinia and Noubir [31] discovered two clear communities. One is the "web programming" languages including JavaScript, PHP, Ruby and CSS. Another form for "System oriented programming" languages, such as C, C++ and Python. Based on the classification result, the authors created a phylogenetic tree for the use of programming languages Github. The authors also ranked top 1000 repositories using different metrics, i.e. percentage of lines of code and PageRank. They believed that HTML is utilized in many repositories with other programming languages, and it is followed by script languages, such as Perl and Makefile.

3 StackOverflow and GitHub: Associations between software development and crowdsourced knowledge

Vasilescu et al. [37] investigated the interaction between StackOverflow actions and the process of development, indicated by the change of code on Github.

Dataset Vasilescu et al. [37] downloaded the list of StackOverflow members and the history of their activities in August 2012. The information including 1,295,622

registered users from July 2008 to August 2012. All of this information are stored in XML format. On the other hand, the data of Github is collected from GHTorrent[12] and the data is stored as MongoDB data dumps. This dataset contains 397,348 users and over 10 million (10,323,714) commits. Most of this information are crawled since July 2011 until April 2012. The authors merge these two datasets following a conservative method, if the computed MD5 hash of the Github user's email address is identical to the MD5 email hash of the StackOverflow user, to prevent the number of false positives. So they totally found 46,967 users who are active on both websites.

Experiment Vasilescu et al. [37] firstly took a macroscopic view to find how Github commits effect the actives on StackOverflow. The applied a "split-and-compare" method on the appropriate statistical testing process and compared multiple distributions. Then they took a intermediate view to understand the commit distribution of users. To satisfy this purpose, they improved their approach by defining a working rhythm. At last, they studied the interaction between activities on Github and StackOverflow.

Result and Conclusion Vasilescu et al. [37] observed that people who commit a lot on Github prefer to answer questions rather than ask questions on StackOverflow. The also found that there is no connection between the working rhythm of Github contributors and their actives on StackOverflow. In addition, they claimed that the active users on StackOverflow are also involved in a great number of contribution on Github and they believed that StackOverflow improves commits on Github.

4 A study of language usage evolution in open source software

Karus and Gall [18] researched the evolution of combined use of programming languages on 22 open source software projects over 12 years.

Dataset Karus and Gall [18] created a dataset including 22 OSS projects and these projects are classified as two categories, desktop type and business(server) type.

Projects, usually used in desktop environments, beyond to desktop type and those projects, providing business function, are regard as business type. For the files of these projects, 45 major file types are identified, such as C, C#, C++.

Experiment Karus and Gall [18] extracted the user information to study their habits and discovered the combination of programming languages in projects. The mainly analyzed three major classes of developers, C/C++ developers, Java developers and XML developers. They also analyzed the first commit made by developers to learn their initial experience. In addition, they studied the commits file types to find out which programming languages are used together and which file types are co-changed.

Result and Conclusion Karus and Gall [18] found that in the 22 OSS projects, the most popular programming language is XML, which is followed by Java and C. They also claimed that the files of the same type easily co-evolve, such as Java file most co-evolve with XML. They believed that XSL is important for the transformation of documents and creation of interfaces. While for developers, although most of them worked on more than programming languages during their studying period, new developers just used a few number of programming languages on their first commit. Based on the usage of programming languages, the authors claimed that developers not only need to utilize multiple programming languages but also should understand different kinds of coding paradigms.

CHAPTER III

Crawling GitHub

1 GitHub

GitHub is a web-based hosted service for Git repositories. Git is a popular open-source version control system. GitHub allows programmers to host remote Git repositories, and also adds a wealth of community-based services. It is a social networking site for programmers. Users can follow others to know their recent activities, and invite other users to collaborate on one repository. GitHub has two kinds of objects, i.e., users and repositories. Figure 1 illustrates their relationships. Each repository is labeled by programming languages, so we also add programming languages in this graph. In this figure, we can notice that there are three different kinds of relationships:

- Users to Users: A user follows other users.
- User to Repository:
  - A user contributes to a repository: A user participates in the development of a repository. The contribution includes adding codes, deleting codes and send commits.
  - A user stars a repository: A user can star a repository. Those users who star a repository are called the stargazers of the repository.
- Repositories to Programming Languages: Repositories are labeled by programming languages based on the number of lines of code. As some repositories are written by using different programming languages, one repository could be labeled by more than one programming language.

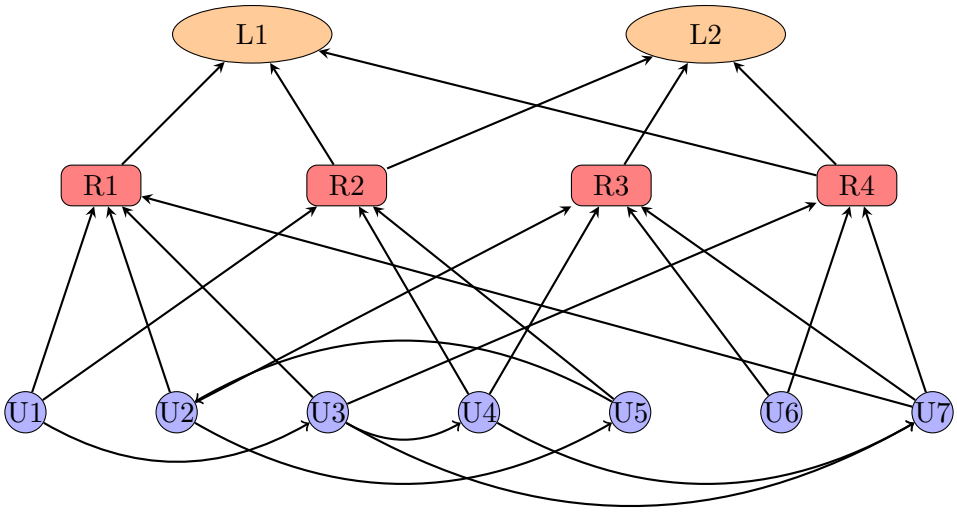


FIGURE 1: GitHub network structure

To analyze GitHub, we first crawl these three kinds of objects. The crawling process and the obtained networks are described in the next subsections.

2 User Network

In the User network, the nodes are the users. There is a link from one node to another if one user follows the other. We crawled all the users and the relationships. The crawling is a challenging task due to the access restriction imposed by GitHub. Each hour an account can send maximal 5000 requests. Each request can return certain data, such as getting the basic information of a user, or a list of its followers. But the list is restricted as well– each request will return one page that contains at most 30 followers. To obtain more followers, additional requests need to be sent to flip through the pages. Because of the limitation on the number of requests and the length of the return list, it is not easy to collect all the data.

More specially, we use the API provided by GitHub to collect the data. First, we need to require authentication by GitHub API and there are three methods to authenticate through GitHub API. The easiest way to authenticate with the GitHub API is by simply utilizing your GitHub login and password via basic authentication.

Then we send requests to GitHub API to collect the information we are interested in. All data is sent and received in JSON format.

2.1 Crawling details

For the user network, we collected all the users and their followers. The crawling process is described below.

- Users: As user id is numbered sequentially[2], we repeatedly send request "https://api.GitHub.com/users?since= **id**;rel='next'"to the service to obtain all the users. In the request, 'since' represents the integer id of the last user you have collected and "rel = 'next'" means getting the next user. For instance, if we let id equals to 136, after sending a request, we receive the information of the user whose id is 137.

  Although the IDs are numbered sequentially, some users were deleted, resulting null return. In this case, we increment the ID until the next user is found. The process of crawling terminates until there is no user returned by the API services.

  Each hour the service returns maximal 5000 different users as the limitation of GitHub API. The information of each user contains login, id and a serious of URLs for its followers, repositories etc.. Figure 2 is an example of the user information we collected from GitHub API.

- Followers: Using the login (the username provided for register) of each user, we repeatedly sent request "https://api.GitHub.com/users/**user login** /followers?page=**page number**" to the API. Each request returns one page of its followers and each page contains 30 followers. To retrieve all the followers, we flip through the pages until the service returns an empty without any followers information.

```
[
  {
    "login": "octocat",
    "id": 1,
    "avatar_url": "https://github.com/images/error/octocat_happy.gif",
    "gravatar_id": "",
    "url": "https://api.github.com/users/octocat",
    "html_url": "https://github.com/octocat",
    "followers_url": "https://api.github.com/users/octocat/followers",
    "following_url": "https://api.github.com/users/octocat/following{/other_user}",
    "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
    "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
    "organizations_url": "https://api.github.com/users/octocat/orgs",
    "repos_url": "https://api.github.com/users/octocat/repos",
    "events_url": "https://api.github.com/users/octocat/events{/privacy}",
    "received_events_url": "https://api.github.com/users/octocat/received_events",
    "type": "User",
    "site_admin": false
  }
]
```


FIGURE 2: Example of user information

2.2 Crawled data

The crawling was done from May 2015 to July 2015. The data of users and followers we collected is summarized in Table 1:

Item	Number
Users	12,007,048
Users have followers	1,190,388
Followers	1,175,466
Relationship between user and followers	6,986,128

TABLE 1: User and follower dataset

Note that in the user graph, the vast majority (90%) of the vertices (users) are isolated, do not connect with any other nodes by the following relationship. For all users, the average number of followers is only 0.58. After removing isolated nodes, the graph only contains 1,668,324 vertices and 6, 986,128 edges. The average degree of followers rises to 4.19. Fig 3 plots the degree distributions of the graph and we noticed that the user degree distribution is close to the power law degree distribution, which is similar to other networks, such as Twitter [20] and Facebook [36]. While, for

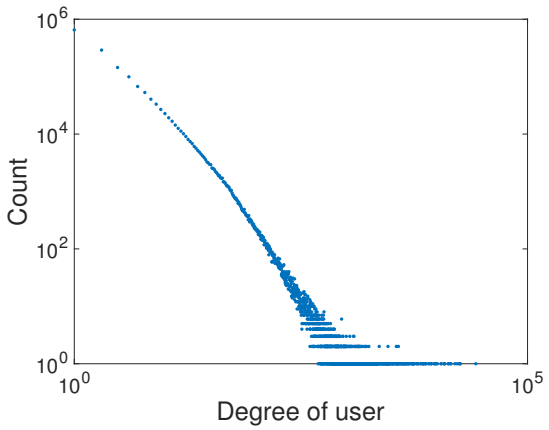


FIGURE 3: Degree distribution of followers

Twitter, the average number of followers is 208 and the number of Facebook is 155.

3 Repository-User Network

There are two types of repository-user networks, each is a bipartite graph. One is the repository-stargazer graph, another is the repository-contributor graph. In the repository-stargazer graph, there are two types of nodes, i.e., repositories and users. There is an edge between a repository and a user if the user gives a star or contribution to the repository. The user is also called a stargazer or contributor.

3.1 Crawling details

- Repository: We scan repositories associated with each user. This is enabled by an HTTP request like "https://api.GitHub.com/users/**user login**/repos?page=**page number**". As the large number of users, there are over 35 millions of repositories but not all of them are useful for us. We just scanned the repository list of each user and downloaded the information of those repositories whose star count is over 500. Because one repository has over 500 stars means this repository is popular and a great number of users follow this repository, these popular repositories can present the features of the whole network in some degree. Figure 4 is an example of the repository. Each repository contains id,

language, which is labeled by GitHub based on the total size of the files that belongs to each programming language, and several links to its attributes, such as its languages, contributors, and stargazers. To obtain details of the language, contributors or stargazers, we need to send requests again.

- Stargazers: As we have the name of repositories, we could collect the stargazers information of repositories. We sent request "https://api.GitHub.com/repos/**repository name**/stargazers?**page number**" to the service and then received the information of stargazers of repositories. And we also used the page number to guarantee that all the stargazers of the repository have been collected.

- Contributors: Same as the crawling of contributors information, we sent request "https://api.GitHub.com/repos/**repository name**/contributors?**page number**" to the service and then received the information of contributors of repositories. As each page only contents 30 contributors, we added the page number to promise all of these contributors information have been crawled.

- Programming Languages: For the repository, we have already collected the programming language of repository labeled by GitHub. While, some repositories may use more than one kind of program language. To get all the programming languages that the repository used, we sent request " https://api.GitHub.com /repos/**repository name**/languages" to API and download all the information including the list of programming languages and the number of lines of code for each programming language. Figure 5 is an example. This repository is written by two programming languages, C and Python. There are 78,769 lines of code is written by C and 7,769 is written by Python in this repository.

3.2 Crawled data

The repository and user data are summarized in Table 2. From the table, we can see that the number of stargazers is over 6 times than the number of contributors. Furthermore, the connection between repository and stargazer is over 50 times than the

```
[
  {
    "id": 1296269,
    "owner": {
      "login": "octocat",
      "id": 1,
      "avatar_url": "https://github.com/images/error/octocat_happy.gif",
      "gravatar_id": "",
      "url": "https://api.github.com/users/octocat",
      "html_url": "https://github.com/octocat",
      "followers_url": "https://api.github.com/users/octocat/followers",
      "following_url": "https://api.github.com/users/octocat/following{/other_user}",
      "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
      "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
      "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
      "organizations_url": "https://api.github.com/users/octocat/orgs",
      "repos_url": "https://api.github.com/users/octocat/repos",
      "events_url": "https://api.github.com/users/octocat/events{/privacy}",
      "received_events_url": "https://api.github.com/users/octocat/received_events",
      "type": "User",
      "site_admin": false
    },
    "name": "Hello-World",
    "full_name": "octocat/Hello-World",
    "description": "This your first repo!",
    "private": false,
    "fork": false,
    "url": "https://api.github.com/repos/octocat/Hello-World",
    "html_url": "https://github.com/octocat/Hello-World",
    "archive_url": "http://api.github.com/repos/octocat/Hello-World/{archive_format}{/
    "assignees_url": "http://api.github.com/repos/octocat/Hello-World/assignees{/user}
    "blobs_url": "http://api.github.com/repos/octocat/Hello-World/git/blobs{/sha}",
    "branches_url": "http://api.github.com/repos/octocat/Hello-World/branches{/branch}
    "clone_url": "https://github.com/octocat/Hello-World.git",
    "collaborators_url": "http://api.github.com/repos/octocat/Hello-World/collaborator
    "comments_url": "http://api.github.com/repos/octocat/Hello-World/comments{/number}
    "commits_url": "http://api.github.com/repos/octocat/Hello-World/commits{/sha}",
    "compare_url": "http://api.github.com/repos/octocat/Hello-World/compare/{base}...{
    "contents_url": "http://api.github.com/repos/octocat/Hello-World/contents/{+path}"
    "contributors_url": "http://api.github.com/repos/octocat/Hello-World/contributors"
    "deployments_url": "http://api.github.com/repos/octocat/Hello-World/deployments",
    "downloads_url": "http://api.github.com/repos/octocat/Hello-World/downloads",
    "events_url": "http://api.github.com/repos/octocat/Hello-World/events",
    "forks_url": "http://api.github.com/repos/octocat/Hello-World/forks",
    "git_commits_url": "http://api.github.com/repos/octocat/Hello-World/git/commits{/s
    "git_refs_url": "http://api.github.com/repos/octocat/Hello-World/git/refs{/sha}",
    "git_tags_url": "http://api.github.com/repos/octocat/Hello-World/git/tags{/sha}",
    "git_url": "git:github.com/octocat/Hello-World.git",
    "hooks_url": "http://api.github.com/repos/octocat/Hello-World/hooks",
    "issue_comment_url": "http://api.github.com/repos/octocat/Hello-World/issues/comme
    "issue_events_url": "http://api.github.com/repos/octocat/Hello-World/issues/events
    "issues_url": "http://api.github.com/repos/octocat/Hello-World/issues{/number}",
    "keys_url": "http://api.github.com/repos/octocat/Hello-World/keys{/key_id}",
    "labels_url": "http://api.github.com/repos/octocat/Hello-World/labels{/name}",
    "languages_url": "http://api.github.com/repos/octocat/Hello-World/languages",
    "merges_url": "http://api.github.com/repos/octocat/Hello-World/merges",
    "milestones_url": "http://api.github.com/repos/octocat/Hello-World/milestones{/num
    "mirror_url": "git:git.example.com/octocat/Hello-World",
    "notifications_url": "http://api.github.com/repos/octocat/Hello-World/notification
    "pulls_url": "http://api.github.com/repos/octocat/Hello-World/pulls{/number}",
    "releases_url": "http://api.github.com/repos/octocat/Hello-World/releases{/id}",
    "ssh_url": "git@github.com:octocat/Hello-World.git",
    "stargazers_url": "http://api.github.com/repos/octocat/Hello-World/stargazers",
    "statuses_url": "http://api.github.com/repos/octocat/Hello-World/statuses/{sha}",
    "subscribers_url": "http://api.github.com/repos/octocat/Hello-World/subscribers",
    "subscription_url": "http://api.github.com/repos/octocat/Hello-World/subscription"
    "svn_url": "https://svn.github.com/octocat/Hello-World",
    "tags_url": "http://api.github.com/repos/octocat/Hello-World/tags",
    "teams_url": "http://api.github.com/repos/octocat/Hello-World/teams",
    "trees_url": "http://api.github.com/repos/octocat/Hello-World/git/trees{/sha}",
    "homepage": "https://github.com",
    "language": null,
    "forks_count": 9,
    "stargazers_count": 80,
    "watchers_count": 80,
    "size": 108,
    "default_branch": "master",
    "open_issues_count": 0,
    "has_issues": true,
    "has_wiki": true,
    "has_pages": false,
    "has_downloads": true,
    "pushed_at": "2011-01-26T19:06:43Z",
    "created_at": "2011-01-26T19:01:12Z",
    "updated_at": "2011-01-26T19:14:43Z",
    "permissions": {
      "admin": false,
      "push": false,
      "pull": true
    }
  }
]
```


FIGURE 4: Example of repository information

```
{
  "C": 78769,
  "Python": 7769
}
```

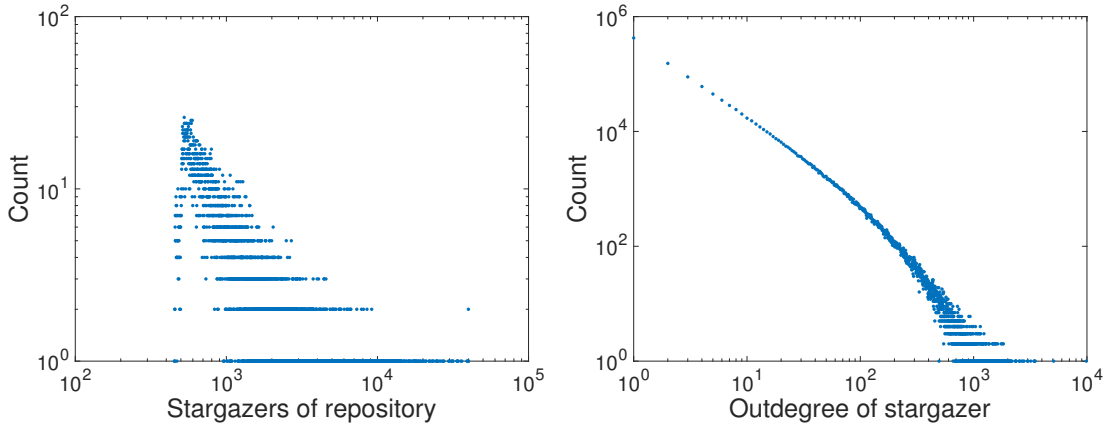

FIGURE 5: Example of language information

Item	Number
Repositories	10,665
Stargazers	1,170,449
Contributors	176,256
Relationship between repositories and stargazers	18,408,518
Relationship between repositories and contributors	354,870
Different programming languages used for these repositories	94

TABLE 2: Repository and user dataset

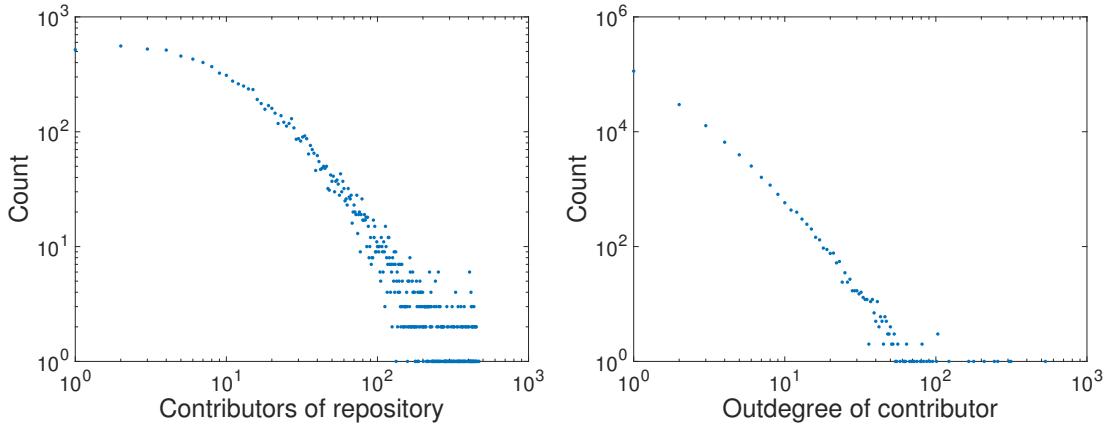
connection between repository and contributor. For the repository-stargazer network, a user gives 15.73 stars in average, and a repository receives 1,726.07 stars in average. Figure 6 shows the distributions of the in-degrees of repositories (a) and out-degrees of the stargazers (b). The in-degree of repositories is the number of stargazers of repositories and the out-degree of stargazers is the number of repositories that users star. From the figures, we realized that the out-degree distribution of stargazers displays the power law distribution. Actually, the in-degree of repositories should also show this, but as we just selected the repository with over 500 stars, we just plotted part of the in-degree distribution of all repositories.

The repository-contributor network is much more sparse. The average in-degree of the repository is 33.27, and the average out-degree of the contributor is 2.01. Here, the in-degree of repositories is the number of contributors of one repository and the out-degree of a contributor is the number of repositories that the user participates in. Figure 7 (a) and (b) separately describes the in-degree distribution of repositories and the out-degree distribution of contributors. From the figure, we noticed that both degree distribution follow the power law distribution.



(a) In-degree distribution of repository (b) Out-degree distribution of stargazers

FIGURE 6: Degree distribution of repository-stargazer network



(a) In-degree distribution of repository (b) Out-degree distribution of contributors

FIGURE 7: Degree distribution of repository-contributor network

Among these 10,665 repositories, there are 94 different programming languages used. Table 3 shows the number of repositories for each programming language. From the table, we can see that the JavaScript is the most popular language (2258), followed by Java (960) and Objective-C (932). Besides, only 34 programming languages have more than ten repositories and there are 487 repositories are unlabelled.

3.3 Sub Networks

In my experiments, we extracted four subnetworks, Python&HTML, Objective-C&C, PHP&CSS, and Java&Ruby, from the repository-stargazer network and repository-contributornetwork. The details of the subnetworks are shown in Table 4. From the table, we noticed that except Java&Ruby dataset, all the other three datasets are unbalanced dataset and Java&Ruby dataset is the largest one, totally including 1842 repositories, 41,034 contributors, and 427,205 stargazers. On the contrary, PHP&CSS dataset is the smallest one with only 779 repositories. We also found that although the repository of PHP&CSS network is much less than Objective-C&C network, PHP&CSS has more contributors than Objective-C&C and the number of stargazers is almost equal. Besides, for the average degree of contributors of four datasets, it is no more than 2. But the average degree of stargazer is at least almost 3 times of the average degree of contributor, which means the repository-stargazer network provides more information and it could obtain better clustering results.

Programming language	Number of repository	Programming language	Number of repository
JavaScript	2866	Objective-C++	3
Java	959	Arduino	3
Objective-C	932	Nginx	3
Python	909	Matlab	3
Ruby	883	D	3
Unlabelled	487	PostScript	2
PHP	461	Perl6	2
C	395	Scheme	2
HTML	356	Batchfile	2
Go	356	F#	2
CSS	318	Vala	2
C++	310	Pascal	2
Swift	234	XML	1
Shell	212	IDL	1
C#	169	MoonScript	1
VimL	128	wisp	1
CoffeeScript	111	Modelica	1
Clojure	78	Yacc	1
Scala	70	NSIS	1
Emacs Lisp	38	PLpgSQL	1
Perl	33	Racket	1
Lua	29	AGS Script	1
Haskell	28	SQLPL	1
Erlang	27	Mirah	1
TeX	23	Awk	1
Rust	22	GLSL	1
Makefile	19	FORTRAN	1
Jupyter Notebook	15	Mathematica	1
TypeScript	14	Nix	1
Groovy	12	PigLatin	1
ActionScript	11	Protocol Buffer	1
R	11	Vue	1
OCaml	11	Nimrod	1
Assembly	10	CMake	1
Elixir	8	Crystal	1
PowerShell	7	KiCad	1
Groff	5	PureBasic	1
Haxe	5	Cuda	1
ApacheConf	5	Frege	1
XSLT	4	nesC	1
Processing	4	Stan	1
Julia	4	Elm	1
Objective-J	4	Arc	1
Kotlin	4	Handlebars	1
Common Lisp	4	Red	1
LiveScript	4	QML	1
Eagle	4	Puppet	1

TABLE 3: Programming Languages

Dataset	Python and HTML	Objective-C and C	PHP and CSS	Java and Ruby
Repository	909 Python 356 HTML	932 Objective-C 395 C	461 PHP 318 CSS	959 Java 883 Ruby
Contributor	35,987	20,893	24,100	41,034
Stargazer	453,451	332,846	326,300	427,205
r-c relation	50,381	29,677	34,104	68,916
r-s relation	2,115,079	2,019,679	1,294,113	2,841,085
av degree-c	1.40	1.42	1.41	1.68
av degree-s	4.66	6.07	3.97	6.65

TABLE 4: Detail of Datasets, r-c relation means relation between repository and contributor, r-s relation means relation between repository and stargazer, av degree-c means average degree of contributors, av degree-s means average degree of stargazers

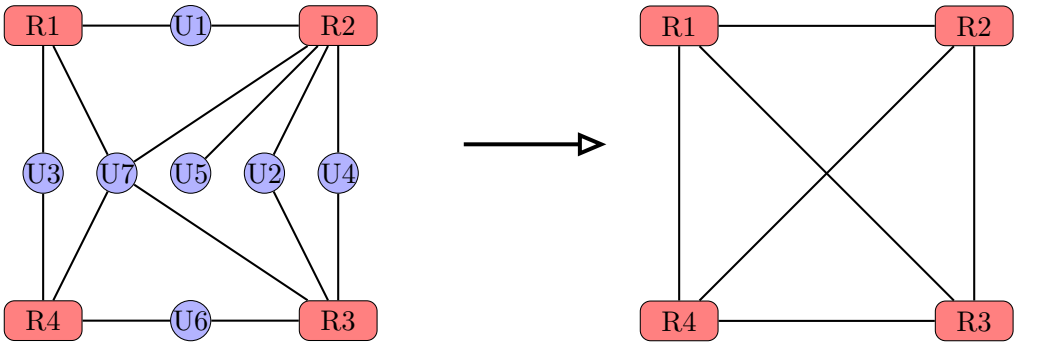
CHAPTER IV

Network Transformation

To cluster the repositories, I applied the heterogeneous-transform homogeneous network clustering [16] on the GitHub dataset that we have crawled. First, I transformed the heterogeneous network constructed by the repositories and users to a homogeneous network which only includes repositories. For the process of transformation, we selected three kinds of weighting schemes, dot product, cosine similarity and Jaccard similarity to represent the similarity between repositories.

We transformed the repository-user networks to repository networks using three weighting schemes, dot product, cosine similarity and Jaccard similarity. In addition, to reduce the effect of users that participate in a large number of repositories, we also applied the inverse document frequency on the repository-user subnetworks and then transformed the new heterogeneous network to the homogeneous one. As Jaccard similarity is only available for the binary object, we just utilized dot product and cosine similarity to complete the transformation.

Based on the relationship between repositories and users(contributors or stargazers), we can construct the repository-user network, a heterogeneous network that includes different kinds of vertices. As the modularity optimization algorithm is only applied to the homogeneous network, which only includes one kind of vertex, we should transform the heterogeneous network into a homogeneous one, which means we should remove the user vertex and form a network only includes repositories(as Figure 8 shown). To satisfy this purpose, we need to determine the similarity between two repositories. In order to represent the similarity, we need to calculate the similarity score between two repositories. For the transformation, we select three



(a) Repository-user network (b) Repository-repository network

FIGURE 8: Heterogeneous network transform to homogeneous network

	U1	U2	U3	U4	U5	U6	U7
R1	1	0	1	0	0	0	1
R2	1	1	0	1	1	0	1
R3	0	1	0	1	0	1	1
R4	0	0	1	0	0	1	1

TABLE 5: Repository-user matrix

different categories of weighting schemes, dot product, cosine similarity and Jaccard similarity. In addition, the repository-user network can also be shown as a matrix. For example, the repository-user network shown in Figure 8 (a) can be represented by the matrix shown in Table 5. Therefore, we can consider each repository as a vector whose element represent the relationship between users and the repositories. If the user is connected to the repository, the value of the element is 1, otherwise is 0. Here we give the definition of the three similarity scores.

1 Dot Product

For a repository-user network, the dot product[33] of two repositories is the number of shared users(stargazers or contributors) between these two repositories. Here is the definition of dot product. For two vectors $A = [A_1, A_2, \dots A_n]$ and $B = [B_1, B_2, \dots B_n]$, the dot product is

$$DotProduct(A, B) = A \cdot B = A_1B_1 + A_2B_2 + \cdots + A_nB_n$$

Here we give an example to explain(see Figure 9 (a) and (b)) how to using the dot product to transform the repository-user network to repository-repository network. If we want to calculate the dot product between R1 and R4, we notice that R1 = [1, 1, 1, 0, 0, 0, 1] and R4 = [0, 0, 1, 0, 0, 1, 1], so the dot product of R1 and R4 is the dot product of these two vectors, then

$$DotProduct(R1, R4) = R1 \cdot R4 = 0 + 0 + 1 + 0 + 0 + 0 + 1 = 2$$

In Figure 9, (a) and (b) displays the dot product transformation for the network and in Table 6 (a) and (b) displays the dot product transformation of the matrix.

2 Cosine Similarity

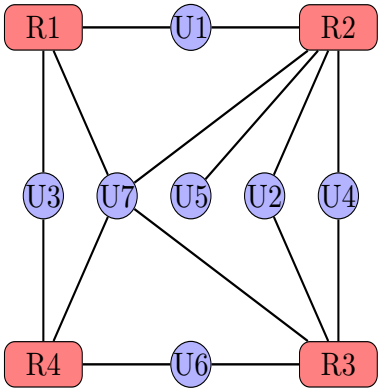
Cosine similarity [15] is measured as the angle between two vectors. As the cosine similarity is easy to explain and simple to calculate for sparse vectors, so it is widely used in clustering [21] and information retrieval [4]. For two vectors A and B, the definition of cosine similarity is

$$Cosine(A, B) = \frac{A \cdot B}{\parallel A \parallel \parallel B \parallel}$$

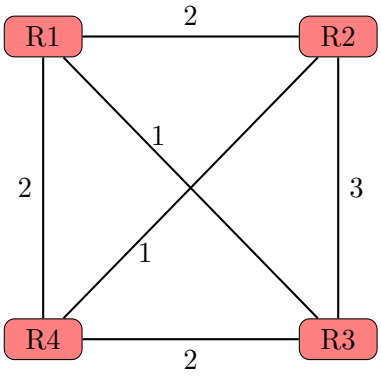
where $\parallel A \parallel$ is the length of vector A.In Figure 9 (a), R1 can be represent as [1, 1, 1, 0, 0, 0, 1] and R4 can be represent as [0, 0, 1, 0, 0, 1, 1], so the cosine similarity between R1 and R4 is

$$Cosine(R1, R4) = \frac{R1 \cdot R4}{\parallel R1 \parallel \parallel R4 \parallel} = \frac{2}{\sqrt{3} \times \sqrt{3}} = \frac{2}{3}$$

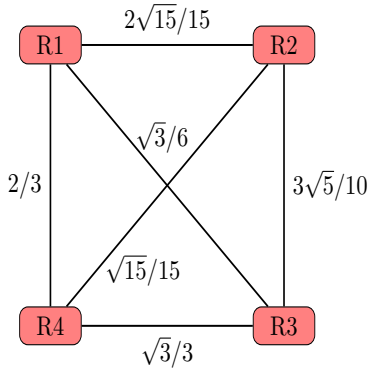
Figure 9 (c) and Table 6 (c) separately shows the results of transformation for the network and the matrix based on cosine similarity.



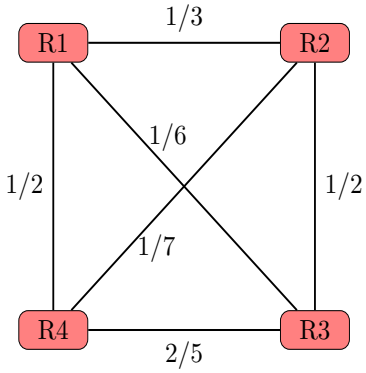
(a) Repository-user network



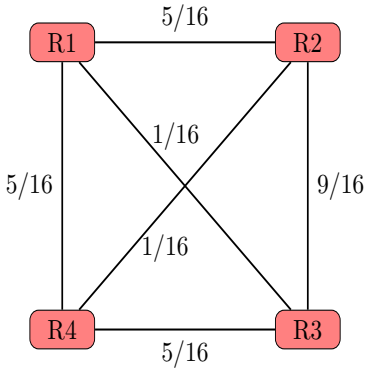
(d) Dop product



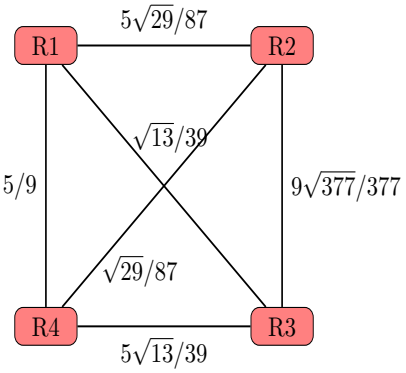
(b) Cosine similarity



(e) Jaccard similarity



(c) Dot Product & IDF



(f) Cosine similarity & IDF

FIGURE 9: Network transformation

	U1	U2	U3	U4	U5	U6	U7
R1	1	0	1	0	0	0	1
R2	1	1	0	1	1	0	1
R3	0	1	0	1	0	1	1
R4	0	0	1	0	0	1	1

(a) Repository-user network

	R1	**R2**	**R3**	**R4**
R1	**0**	**2**	**1**	**2**
R2	**2**	**0**	**3**	**1**
R3	**1**	**3**	**0**	**2**
R4	**2**	**1**	**2**	**0**

(b) Dop product transformation

	R1	R2	R3	R4
R1	0	$2\sqrt{15}/15$	$\sqrt{3}/6$	2/3
R2	$2\sqrt{15}/15$	0	$3\sqrt{5}/10$	$\sqrt{15}/15$
R3	$\sqrt{3}/6$	$3\sqrt{5}/10$	0	$\sqrt{3}/3$
R4	2/3	0	$\sqrt{15}/15$	0

(c) Cosine similarity transformation

	R1	**R2**	**R3**	**R4**
R1	**0**	**1/3**	**1/6**	**1/2**
R2	**1/3**	**0**	**1/2**	**1/7**
R3	**1/6**	**1/2**	**0**	**2/5**
R4	**1/2**	**1/7**	**2/5**	**0**

(d) Jaccard similarity transformation

	R1	**R2**	**R3**	**R4**
R1	**0**	**5/16**	**1/16**	**5/16**
R2	**5/16**	**0**	**9/16**	**1/16**
R3	**1/16**	**9/16**	**0**	**5/16**
R4	**5/16**	**1/16**	**5/16**	**0**

(e) Dot Product & IDF

	R1	R2	R3	R4
R1	0	$5\sqrt{29}/87$	$\sqrt{13}/39$	5/9
R2	$5\sqrt{29}/87$	0	$9\sqrt{377}/377$	$\sqrt{29}/87$
R3	$\sqrt{13}/39$	$9\sqrt{377}/377$	0	$5\sqrt{13}/39$
R4	5/9	$\sqrt{29}/87$	$5\sqrt{13}/39$	0

(f) Cosine similarity & IDF

TABLE 6: Matrix transformation

3 Jaccard Similarity

Jaccard similarity [34] is usually used to deal with data objects which have binary attributes. In our network, we consider the user as the attributes for each repository, so there are just two kinds of situation. One is the user contribute or star the repository. Another is that there is no connection between the repository and user. So each repository can be represented as a vector with binary value for each user. Then for two vectors A and B, the definition of Jaccard similarity is as follows:

$$Jaccard(A, B) = \frac{A \cdot B}{\parallel A \parallel^2 + \parallel B \parallel^2 - A \cdot B}$$

As shown in table 6(a), R1 is equal to [1, 1, 1, 0, 0, 0, 1] and R4 is equal to [0, 0, 1, 0, 0, 1, 1], so the Jaccard similarity between R1 and R4 is

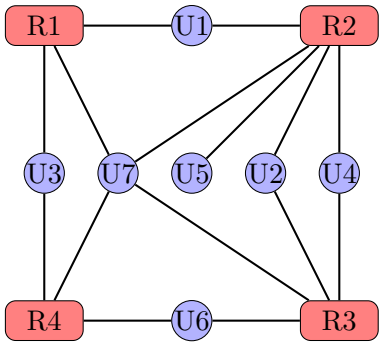
$$Jaccard(R1, R4) = \frac{R1 \cdot R4}{\parallel R1 \parallel^2 + \parallel R4 \parallel^2 - R1 \cdot R4} = \frac{2}{3 + 3 - 2} = \frac{1}{2}$$

Figure 9 (d) and table 6 (d) separately displays the consequences of network and matrix after the transformation by applying Jaccard similarity.

4 Inverse Document Frequency Transformation

In our dataset, some users may participate(contribute or star) in a large number of repositories, but their effect to repositories are same as those users who just participate in a few repositories. For example, from the matrix shown in Figure 10, we noticed that U5 just participate one repository R2 and U7 participate all of the repositories. But the effect of both U5 and U1 to R2 is 1, which is unreasonable. Following the IDF(Inverse Document Frequency) heuristic in information retrieval, where popular words are discredited inversely proportional to their document frequencies, we also suggest to reduce a user's weight inversely proportional to its stars or contributs she gives.

Inverse document frequency(IDF) is proposed by Sparck Jones [32] in 1972. It is used to evaluate how important a term is to a document. As a term which displays in a large number of documents is not a good discriminator, so it should be given



(a) Repository-user network

	U1	U2	U3	U4	U5	U6	U7
R1	1	0	1	0	0	0	1
R2	1	1	0	1	1	0	1
R3	0	1	0	1	0	1	1
R4	0	0	1	0	0	1	1

(b) Repository-user matrix

FIGURE 10: Example of IDF transform

	U1	U2	U3	U4	U5	U6	U7
R1	1	0	1	0	0	0	1
R2	1	1	0	1	1	0	1
R3	0	1	0	1	0	1	1
R4	0	0	1	0	0	1	1

(a) Repository-user matrix

	U1	U2	U3	U4	U5	U6	U7
R1	1/2	0	1/2	0	0	0	1/4
R2	1/2	1/2	0	1/2	1	0	1/4
R3	0	1/2	0	1/ 2	0	1/2	1/4
R4	0	0	1/2	0	0	1/2	1/4

(b) Repository-user matrix after IDF

TABLE 7: Example of IDF transform

less weight than those which just occur in a few documents. In our dataset, due to some user join many repositories, we should also set less weight for these users. So we applied the same idea for both repository-stargazer and repository-contributor network to attenuate the effect of users that occur too often in the repositories. Suppose the weight of each user is 1, so the formula we used to calculate the IDF of a user:

$$Idf_u = \frac{1}{d_u} \tag{1}$$

where d_u is the number of repositories that user u contribute to or star. For instance(see Figure 10), U1 is connected to two repositories, so $Idf_{U1} = \frac{1}{d_{U1}} = 1/2$. Like this, we can calculate the IDF for the other users and the result is shown in Figure 7(b). Compared with the table shown in Table 7(a), we see that except the weight of U5 is still 1, all the weight of other users decrease.

As Jaccard similarity is only available for binary vector, we applied dot prod-

uct and cosine similarity on the heterogeneous network which has been transformed by inverse document frequency. Figure 9 (e) and (f) separately show the result of repository-user network transformed by dot product & IDF and cosine similarity & IDF. Table 6(e) and (f) show the result of the transformation of the matrix. Compared with the result that the network only transformed by dot product and cosine similarity, we found that the process of inverse document frequency increase the ratio between two weights, which means the distance between two repositories is larger.

CHAPTER V

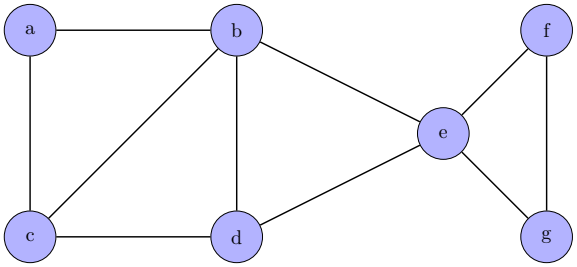
Clustering Networks

Apply the weighting schemes introduced in chapter IV, we obtain the homogeneous repository-repository network. For the homogeneous network clustering, we selected two modularity maximization optimization algorithms. One is greedy modularity maximization optimization algorithm [25] and the other one is spectral modularity maximization optimization algorithm[26]. To evaluate the clustering, we selected two measures, one is F-measure [38] and another is index(RI) [30].In this section, we firstly introduce the definition of modularity and then explain the theory of these two algorithms with examples. At last, we applied these two algorithms on the subnetwork we introduced in section 3.3 and evaluate to find the best weighting scheme to reflect the relation between repositories.

1 Definition of Modularity

The definition of modularity is firstly proposed by Newman and Girvan [27] in 2004 and it is based on a serious of the previous measure of assortative mixing appropriate to the various mixing types [24]. Modularity is a measure used to scale the quality of the particular community structure of a network. It can be defined as the fraction of edges fall within communities minus the expected fraction of edges fall within communities of same size network which is randomly distributed. Therefore, for an unweighted and undirected network G(V,E), V is the set of vertices and $|V| = n$ and E is the set of edges and $|E| = m$. The modularity is

$$Q = \frac{1}{2m} \sum_{i,j \in V} (A_{ij} - P_{ij}) s(c_i, c_j) \tag{1}$$



(a) Example network

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

(b) Adjacency matrix

FIGURE 11: Example of adjacency matrix

where A_{ij} is the element of the adjacency matrix A and P_{ij} represents the expected number of edges between vertices i and j under the random network of the same size. In addition, c_i is the community of vertex i belong to, and $s(c_i, c_j)$ represents whether vertex i and j are in the same community and its value is defined as follows

$$s(c_i, c_j) = \begin{cases} 1 & \text{i,j belongs to same community} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

For the adjacency matrix A, it is a square matrix used to represent a finite graph and the element A_{ij} indicates whether pairs of vertices i and j are adjacent or not in the graph. The value of A_{ij} is defined as follows

$$A_{ij} = \begin{cases} 1 & \text{if vertices i and j are connected} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Here is an example, the network is shown in Figure 11(a) can be represented by the adjacency matrix shown in figure 11(b).

To calculate the expected number of edges between vertices, we suppose each edge in the network is cut to two stubs [9]. So if we want to form an edge between tow vertices i and j, we just need to join two stubs, separately connect with vertex i and vertex j. If k_i is the degree of vertex i and m represents the number of edges in the graph, the probability of picking a random stub connected with vertex i is $p_i = \frac{k_i}{2m}$.So

the probability of a connection between vertex i and vertex j is $p_i p_j = \frac{k_i k_j}{4m^2}$. Then we get the expected number of edges between i and j is $P_{ij} = 2mp_i p_j = \frac{k_i k_j}{2m}$. For example, for the network shown in Figure 11 (a), we notice that the degree of vertex b is $k_b = 4$ and the degree of vertex d is $k_b = 3$. And in this network, there are totally 10 edges, so the expected number of edges between vertex b and vertex d is: $\frac{k_b k_e}{2m} = \frac{4*3}{2*10} = \frac{3}{5}$.

So we finally get the mathematical definition of modularity

$$Q = \frac{1}{2m} \sum_{ij \in V} (A_{ij} - \frac{k_i k_j}{2m}) s(c_i, c_j) \tag{4}$$

Here we notice that the value of modularity is only depended on the vertices belong to the same communities. The highest value of modularity is 1, which indicates networks have strong community structure. While, the value can also be negative and this implies that there are more edges between communities than edges fall within communities. In practice, the value of modularity is usually fallen between about 0.3 and 0,7. Although the high value of modularity means strong community structure for the network, it is rare to get a high one.

2 Modularity Maximization

As we have mentioned in the last section, the high value of modularity implies that the division strategy for the network has high quality or at least the division choice is a good one to get strong community structure, so why not simply optimize modularity over all possible divisions to find the best one. The problem is that true optimization of modularity is extremely costly. If we suppose S(n,k) represents the number methods to cluster n vertices into k communities, we can have the iterative definition for S(n,k),

$$S(n,k) = S(n-1,k-1) + kS(n-1,k) \tag{5}$$

where S(n, 1) = S(n,n) = 1. The sum of $S(n,k)$ cannot be transformed to or represented as any closed form, but we can observe that $S(n,1) + S(n,2) = 1 + 2^{n-1} - 1 = 2^{n-1}$ for all $n > 1$ [25]. So,the growth of the sum of $S(n,k)$ must be at least exponential in n . Accomplish a complete search of all possible division strategies to obtain

the optimal value of modularity would, therefore, take at least an exponential amount of time, which means the optimization of maximum modularity is NP-hard problem [6]. However, there are various available approximate optimization methods, such as the greedy modularity maximization optimization algorithm and the spectral modularity maximization optimization algorithm, which will be introduced in the follow sections.

2.1 Greedy Modularity Maximization Optimization Algorithm

The greedy modularity maximization optimization algorithm is proposed by Newman [25]. This greedy algorithm is an agglomerative hierarchical clustering method. This algorithm supposes each vertex in the network is a community at the first step, so there are n communities in total. And the purpose of this algorithm is to get the largest increase in modularity for each merge of clusters. Therefore, for each step, the algorithm repeatedly merges two communities together and selects the pair of communities that leading to the most increase or least decrease of modularity to combine. Suppose we merge community r and community s together and using t to represent the combined community. As the merge does not change the modularity of other communities, the change of modularity ΔQ_{rs} can be calculated as

$$\begin{aligned} \Delta Q_{rs} &= Q_t - Q_r - Q_s \\ &= e_{rs} + e_{sr} - 2a_r a_s \\ &= 2(e_{rs} - a_r a_s) \end{aligned} \tag{6}$$

where e_{rs} is one-half of the fraction of edges in the network that connects vertices in community r to community s. So if $\delta(c_i, r) = 1$ represents vertex i belong to community r, otherwise $\delta(c_i, r) = 0$, we have

$$e_{rs} = \frac{1}{2m} \sum_{ij} A_{ij} \delta(c_i, r) \delta(c_j, s) \tag{7}$$

and $a_r = \sum_s e_{rs}$ is the fraction of all ends of edges that are attached to vertices in community r, so

$$a_r = \frac{1}{2m} \sum_{i} k_i \delta(c_i, r) \tag{8}$$

The algorithm is terminated when there is just one community left, so there is totally $n-1$ merge steps, where n is the number of vertices in the network. Thus the general form of greedy modularity maximization algorithm is shown as follows:

Algorithm 1: Greedy Modularity Maximization Algorithm

Consider each vertex as a community;

repeat

Repeatedly merge two communities together and calculate the change of modularity;

Merge the pair of communities leading to most increase or least decrease in modularity;

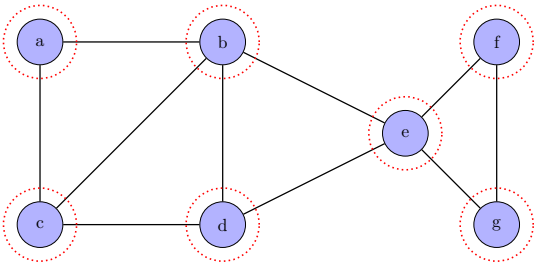
Add the change of modularity to the total modularity;

until *there is just one community left*;

Select the state with highest total modularity as the final division;

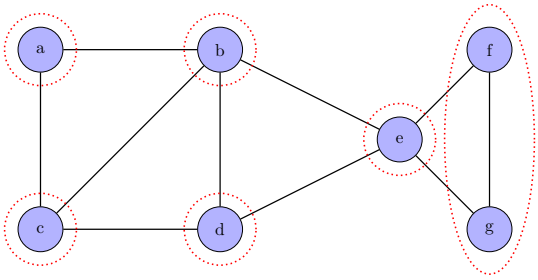
Figure 12 is an example, which explains how the greedy modularity maximization algorithm works. We see that the network includes 7 vertices, so for the initial step, there are 7 communities. At this state, the modularity of the network is $Q_0 = \frac{1}{2m}\sum_{ij\in V}(A_{ij} - \frac{k_i k_j}{2m})s(c_i, c_j) = -0.155$. Then we merged each pair of vertices together and found that when we merged f and g together, we got the most increase of modularity, which is equal to $\Delta Q_{fg} = 2(e_{fg} - a_f a_g) = 2 \times (\frac{1}{20} - \frac{2}{20} \times \frac{2}{20}) = 0.08$. So we put regard f and g as one community.For now, there are 6 communities left and the total modularity is $Q_1 = Q_0 + max(\Delta Q) = Q_0 + \Delta Q_{fg} = -0.155 + 0.08 = -0.075$. Then we repeat the process of merging(Step 2 to 5) until all of the vertices belong to the same community, as the Step 6 shown. We can clearly see that at step 5 we get the highest modularity, $Q_5 = 0.28$, so the state of step 5 is the final division strategy, which means the whole network is divided into two communities. One community is constructed a, b,c,d and e,f, g belong to the other one.

The whole process of merging can also be shown as a dendrogram, a tree describes the order of merging. Figure 13 shows the dendrogram tree of the example. We can see that firstly we merge f and g together, and then put e into the new community.



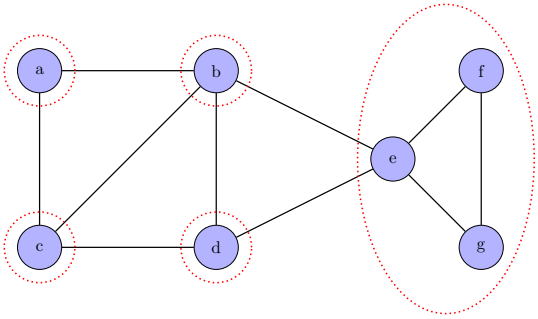
(a) Initial state

$Q_0 = -0.155$



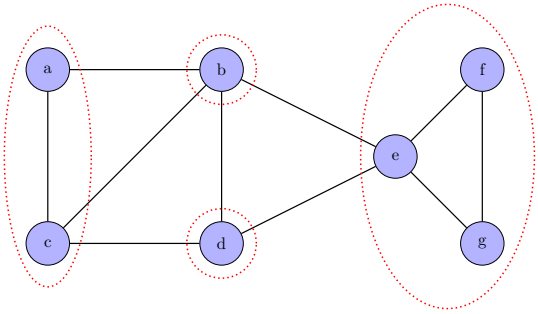
(b) Step1: merge f and g

$Q_1 = -0.075$



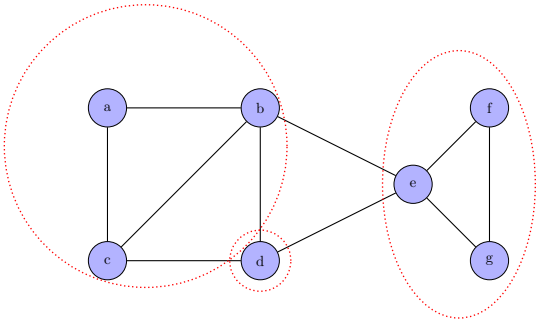
(c) Step2:merge e and f,g

$Q_2 = 0.045$



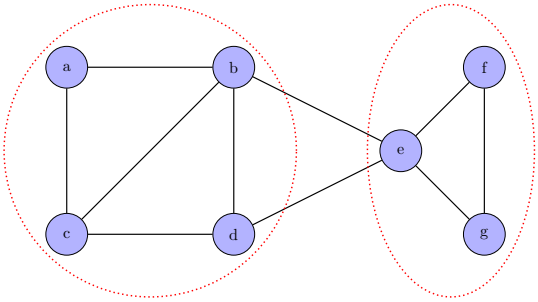
(d) Step3:merge a and c

$Q_3 = 0.115$



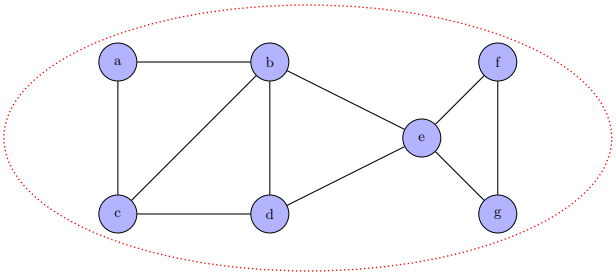
(e) Step4:merge c and a,b

$Q_4 = 0.215$



(f) Step5:merge d and a,b,c

$Q_5 = 0.28$



(g) Step6:merge a,b,c,d and e,f,g

$Q_6 = 0.155$

FIGURE 12: Example of greedy modularity algorithm

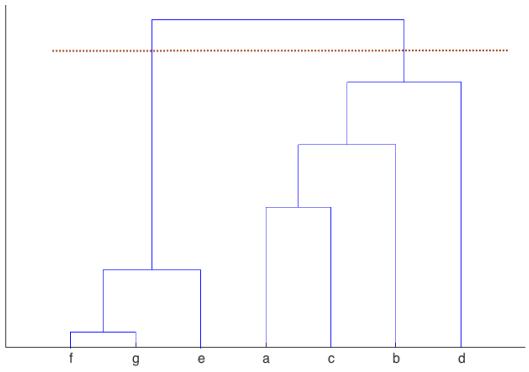


FIGURE 13: Dendrogram result of greedy algorithm

Next, we orderly merge a, c ,b and d to form one community. At last, we merge these two large communities together. This process is same as the example we have shown in Figure 12.In addition, cutting through the dendrogram can get the division strategy of the network. As Figure 13 shown, if we divide the network as the red dot line, the whole network into two communities. The result is also same as the one we have got when step 5.

2.2 Spectral Modularity Maximization Optimization Algorithm

The spectral modularity maximization optimization algorithm is proposed by Newman [26]. This algorithm implies that modularity can be represented as the eigenvectors of the modularity matrix, a characteristic matrix for the network, and applies a spectral method on this expression to obtain a high-quality community detection for the network. For each step, the algorithm divides the network into two communities based on the sign of the leading eigenvector and then repeats the process on the two communities. So first we introduce how to divide a network into two communities. This algorithm applies a different approach to define the modularity. Unlike the original one, Newman [26] supposes dividing a network into two communities and

represents such a division by the quantities as follow:

$$s_i = \begin{cases} +1 & \text{if vertex i belongs to community 1} \\ -1 & \text{if vertex i belongs to community 2} \end{cases} \tag{9}$$

So if vertex i and j are in the same community, $\frac{1}{2}(s_i s_j + 1)$ is 1, otherwise the value is 0. Based on this, the modularity can be represented as follows

$$Q = \frac{1}{2m}(A_{ij} - \frac{k_i k_j}{2m})(s_i s_j + 1) = \frac{1}{4m}(A_{ij} - \frac{k_i k_j}{2m})s_i s_j \tag{10}$$

This equation can be written in matrix form as

$$Q = \frac{1}{4m} s^T B s \tag{11}$$

where **s** is the column vector with s_i as the elements. And B is called the modularity matrix, a real symmetric matrix whose elements are

$$B_{ij} = A_{ij} - \frac{k_i k_j}{2m} \tag{12}$$

The goal of the algorithm is to find a good division of the network that gets the highest modularity, which means we need to find the maximum value of equation 11 for a given modularity matrix B. To satisfy this purpose, let $u_i (i = 1, \cdots n)$ be eigenvectors of B with eigenvalue β_i for vector u_i(Assume $\beta_1 \geq \beta_2 \geq \beta_2 \geq \cdots \geq \beta_k$). So **s** can be written as $s = \sum_i^n a_i u_i$, where $a_i = u_i^T \cdot s$. Now the equation 11 can be transformed as follows

$$\begin{aligned} Q &= \frac{1}{4m} s^T B s \\ &= \frac{1}{4m} (\sum_i a_i u_i^T) B (\sum_j a_j u_j) \\ &= \frac{1}{4m} (\sum_i \sum_j a_i a_j u_i^T B u_j) \end{aligned} \tag{13}$$

As u_i and β_i are the corresponding eigenvector and eigenvalue of B, we have $Bu_i = \beta_i u_i$. Besides, due to the distinct eigenvectors of the symmetric matrix are orthogonal. Therefor, if $i \neq j$, the value of $u_i^T B u_j$ is 0. Then the formulation of modularity can be simplified as follows

$$Q = \frac{1}{4m} (\sum_i (u_i^T s)^2 \beta_i) \tag{14}$$

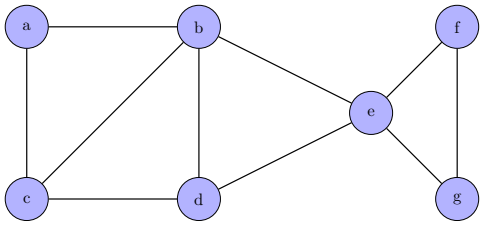


FIGURE 14: Example for spectral modularity optimization algorithm

So, to get the highest modularity by choosing an appropriate clustering of the network, we hope to find a $\mathbf{s}$ to gain as much value as possible for the sum in equation 14. If there are no other restrictions, this problem could be solved easily by choosing the $\mathbf{s}$, which is parallel to the u_1, which is the leading eigenvector of the modularity matrix [23]. This promise the largest eigenvalue β_1 is involved in the sum and the other terms in the equation are automatically zero, because the eigenvectors of the symmetric matrix are orthogonal to each other. While, unfortunately, there is a restriction that the value of s_i should be 1 or -1, which means $\mathbf{s}$ would not be easily selected parallel to u_1. So the best we can do is to make $\mathbf{s}$ as close parallel as possible to u_1. In others words, we should keep the dot product $u_1^T \cdot s$ be the highest value. To maximize the dot product, we should let $s_i = 1$ if the corresponding element of u_1 is positive, otherwise $s_i = -1$. Then based on the sign of s_i, we would divide the whole network into two communities. The positive value is one community and the negative ones compose another one. Here is an example displays how this algorithm divides a network into two communities.For the network shown in figure 14, based on equation 12 we constructed the modularity matrix B, as shown in Figure 15. Then based on the modularity matrix, we calculated the leading vector(see Figure 16) of this modularity matrix. At last, based on the sign of leading vector, we can divide the network into two parts, which is shown in figure 17. We can see that the clustering result is same as the one we have obtained by the greedy modularity maximization optimization algorithm.

However, in the real world, most of the networks contain more than two communities, so we extend the spectral modularity maximization optimization algorithm to find appropriate divisions of the network into several communities. To address this

$$B = \frac{1}{20}\begin{pmatrix} -4 & 12 & 14 & -6 & -8 & -4 & -4 \\ 12 & -16 & 8 & 8 & 4 & -8 & -8 \\ 14 & 8 & -9 & 11 & -12 & -6 & -6 \\ -6 & 8 & 11 & -9 & 8 & -6 & -6 \\ -8 & 4 & -12 & 8 & -16 & 12 & 12 \\ -4 & -8 & -6 & -6 & 12 & -4 & 16 \\ -4 & -8 & -6 & -6 & 12 & 16 & -4 \end{pmatrix}$$

FIGURE 15: Modularity matrix

$$Leading\ eigenvector = \begin{pmatrix} -0.37 \\ -0.30 \\ -0.43 \\ -0.17 \\ 0.32 \\ 0.48 \\ 0.48 \end{pmatrix}$$

FIGURE 16: Leading eigenvector

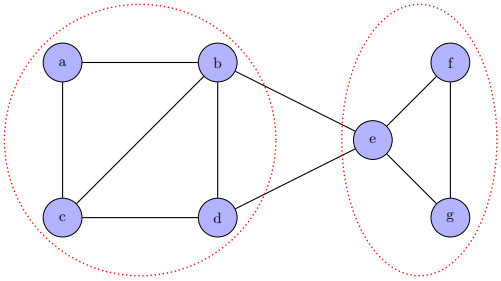


FIGURE 17: Result of spectral modularity optimization algorithm

problem, we first divide the network into two subnetworks and then repeatedly apply the algorithm on the subnetwork divided by the last step and compute the additional modularity $\triangle Q$ for each division. Suppose dividing a network n of size l_n into two subnetworks, the additional modularity to the whole modularity is

$$\triangle Q = \frac{1}{2m}[\frac{1}{2}\sum_{i,j\in n} B_{ij}(s_i s_j + 1) - \sum_{i,j\in n} B_{ij}] \tag{15}$$

As $\sum_{i,j\in c} B_{ij} = \sum_{i,j\in c} s_i s_j \delta_{ij} \sum_{k\in c} B_{ik}$ and let $\omega_{ij} = 1$ if i = j, otherwise is 0, the equation can be simplified as follows

$$\begin{aligned} \triangle Q &= \frac{1}{2m}[\frac{1}{2}\sum_{i,j\in n} B_{ij}(s_i s_j + 1) - \sum_{i,j\in n} B_{ij}] \\ &= \frac{1}{4m}[\frac{1}{2}\sum_{i,j\in c} B_{ij} s_i s_j - \sum_{i,j\in c} B_{ij}] \\ &= \frac{1}{4m}\sum_{i,j\in c}[B_{ij} - \omega_{ij}\sum_{k\in c} B_{ik}]s_i s_j \\ &= \frac{1}{4m} s^T B^{(n)} s \end{aligned} \tag{16}$$

where $B^{(n)}$ is the matrix of size $l_n \times l_n$ with elements which is indexed by the label i,j of nodes in network n and having weights

$$B_{ij}^{(g)} = B_{ij} - \omega_{ij} \sum_{k \in g} B_{ik} \tag{17}$$

We noticed that equation 16 has the same form of equation 11, so we also apply the spectral method to maximize $\triangle Q$ of the generalized modularity matrix. If there is no positive value of $\triangle Q$, the subgroup is indivisible. Thus the algorithm is shown as Algorithm 2:

Algorithm 2: Spectral Modularity Maximization Algorithm

Construct the modularity matrix for the network;

repeat

Find the leading eigenvalue and corresponding eigenvector of the modularity matrix;

Add the change of modularity to the total modularity;

Divide the network into two communities based on the signs of the elements of the eigenvector;

Construct the modularity matrix for the subnetwork;

until *find the split make no positive contribution to the total modularity*;

3 Evaluation measures

We use two categories of evaluation measure. One is F-measure [38], which is based on the cluster matching. Another is Rand Index(RI) [30], which is based on the pair counting. We use programming language labeled by GitHub as the ground truth.

3.1 F-measure

F-measure [38] is usually used to measure the similarity between to partitions. Its value depends on the combination of precision and recall. Precision is the percentage

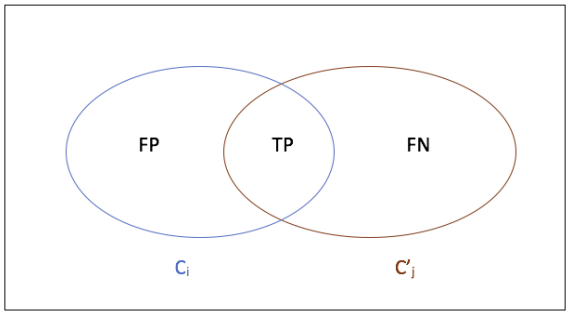


FIGURE 18: F-measure

of selected items that are correct and recall represents the percentage of correct items that are selected. Suppose Figure 18 is a network, $C = \{C_1, \cdots C_k\}$ is the ground truth for network and $C' = \{C'_1, \cdots C'_k\}$ represents a clustering for the network. So the precision for each pair of C_i and C'_j

$$Precision = \frac{TP}{TP + FP} = \frac{|C_i \cap C'_j|}{|C_i|} \tag{18}$$

and recall is

$$Recall = \frac{TP}{TP + FN} = \frac{|C_i \cap C'_j|}{|C'_j|} \tag{19}$$

Then we get the F-measure for C_i and C'_j is

$$\text{F-measure}(C_i, C'_j) = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot |C_i \cap C'_j|}{|C_i + C'_j|} \tag{20}$$

So the overall F-measure is the weighted sum of the maximum F-measures for clusters C:

$$\text{F-measure}(C, C') = \frac{1}{|V|} \sum_{c_i \in C} |c_i| \max_{c'_j \in C'} \frac{2|c_i \cap c'_j|}{|c_i| + |c'_j|} \tag{21}$$

Here is an example(see Figure 19) explain how to calculate the F-measure. In the example, there are 10 vertices. The blue circles represent the clusters of ground truth and the red circles are the clusters we have got. So we have C : $c_1\{a, b, c, d, e\}$, $c_2\{f, g, h, i, k\}$ and C': $c'_1\{a, b, c\}$, $c'_2\{d, e, f\}$, $c'_3\{g, h, i, k\}$. So based on equation 20, we calculate the F-measure of each pair of C and C' and result is shown as follows

$$\frac{2|c_1 \cap c'_1|}{|c_1| + |c'_1|} = \frac{2 \times 3}{5 + 3} = \frac{3}{4} \qquad \frac{2|c_1 \cap c'_2|}{|c_1| + |c'_2|} = \frac{2 \times 2}{5 + 3} = \frac{1}{2} \qquad \frac{2|c_1 \cap c'_4|}{|c_1| + |c'_4|} = \frac{2 \times 0}{5 + 4} = 0$$

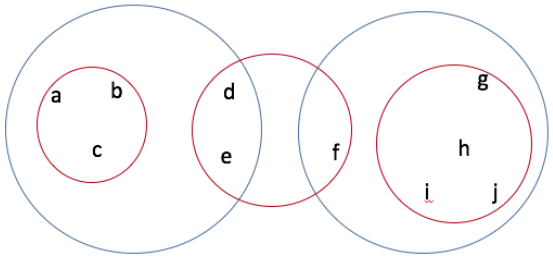


FIGURE 19: Example of F-measure

$$\frac{2|c_2 \cap c_1'|}{|c_2| + |c_1'|} = \frac{2 \times 0}{5+3} = 0 \qquad \frac{2|c_2 \cap c_2'|}{|c_2| + |c_2'|} = \frac{2 \times 1}{5+3} = \frac{1}{4} \qquad \frac{2|c_2 \cap c_3'|}{|c_2| + |c_3'|} = \frac{2 \times 4}{5+4} = \frac{8}{9}$$

So the overall F-measure of clustering C' is

$$\text{F-measure}(C, C') = \frac{1}{10}(5 \times \frac{3}{4} + 5 \times \frac{8}{9}) \approx 0.82$$

3.2 Rand Index

Rand Index [30] is also a measure used to evaluate the similarity between two clusterings. Unlike F-measure, rand index considers the pair of vertices. It represents the fraction of the number of vertex pair which is clustered in the same ways in both clusterings to the total number of pairs. So errors only happen under two situations. One is two vertices belonging to the same community are assigned to different communities after clustering and another is two vertices belonging to different communities are assigned to the same community.

		Ground Truth	
		$C(v_i) = C(v_j)$	$C(v_i) \neq C(v_j)$
Clustering Result	$C'(v_i) = C'(v_j)$	a_{11}	a_{10}
	$C'(v_i) \neq C'(v_j)$	a_{01}	a_{00}

TABLE 8: Rand index

Suppose C represents the ground truth of a network and C' is a clustering for the same network. As Table 8 shown, a_{11} represent the number of pair of vertices that

are in the same clusters in both the ground truth and the partition. a_{10} indicates the count of pairs of vertices that are in the different communities of the ground truth but in the same community for in partition C'. a_{01} denote the number of pairs of vertices that in the same community of ground truth which are in different communities in C'. And a_{00} be the number of pairs of vertices which are in different clusters for both ground truth C and clusters C'. So $A = a_{11} + a_{01} + a_{10} + a_{00}$ is the total number of pairs of vertices in the network. So the rand index is

$$Rand\ Index = \frac{a_{11} + a_{00}}{a_{11} + a_{01} + a_{10} + a_{00}} \tag{22}$$

4 Clustering Results

In this subsection, we introduced the experiment we have done to study the interaction between different weighting schemes and clustering algorithms. In my experiments, we extracted four subnetworks(Python & HTML, Objective-C & C, PHP & CSS and Java & Ruby), which only includes two categories of programming language repository, from both repository-contributor network and repository-stargazer network. Then we transformed the heterogeneous subnetwork into the homogeneous network, which only includes repositories, by using three weighting schemes, dot product, cosine similarity and Jaccard similarity. In addition, we also applied the inverse document frequency on the original heterogeneous network to reduce the effect of those users, who participate too many repositories, and then transformed the new network by dot product and cosine similarity, because Jaccard similarity is only available for binary one. At last, we separately applied the greedy modularity optimization algorithm and spectral modularity optimization algorithm to cluster the homogeneous network. As we have already known that each dataset only includes two kinds of repositories, we just cluster the network into two communities and evaluate the performance of different weighing schemes by F-measure and rand Index with ground truth. In addition, we also analyzed those repositories that are clustered into the wrong communities to find out the reason.

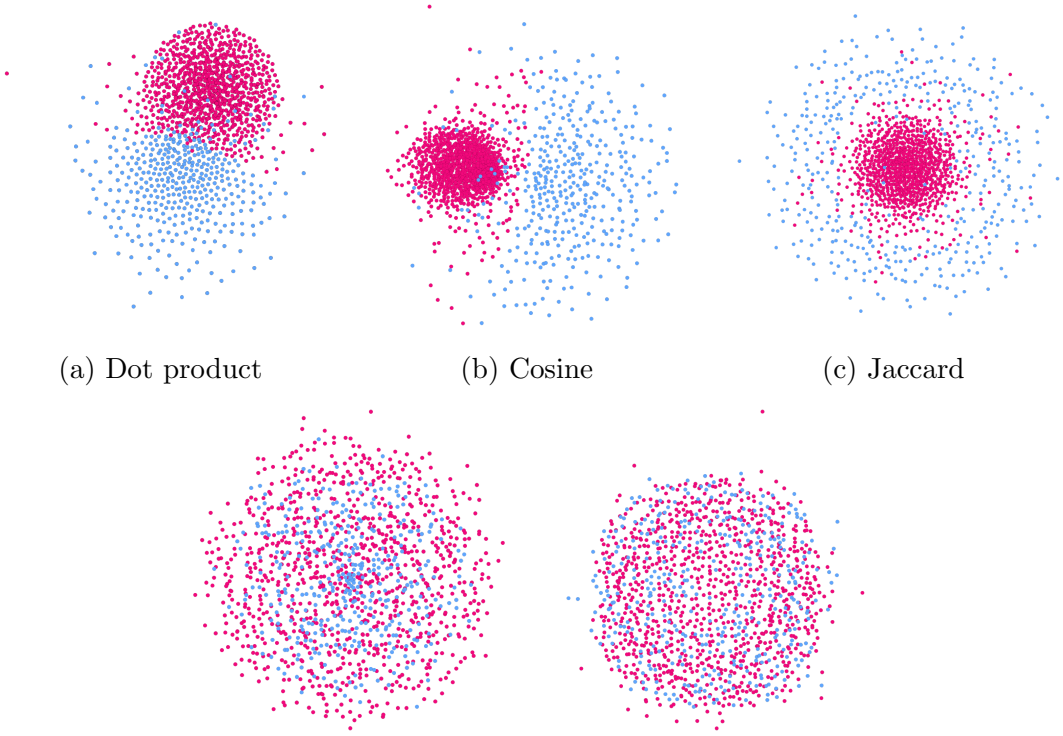
In my experiments, I have implemented the process of data crawling, extraction

of subnetworks, network transformation, and network clustering. All of the code is written in Python. Except the implementation of greedy modularity maximization optimization algorithm, I used the function community_fastgreedy of igraph [7], all the other code is original.

4.1 Visualization of the Original Clusters

Before running the clustering algorithm, we visualized the clusters using labeled data. First, we transformed the repository-user networks to repository networks using three weighting schemes, dot product, cosine similarity and Jaccard similarity. In addition, to reduce the effect of users that participate in a large number of repositories, we also applied the inverse document frequency on the repository-user subnetworks and then transformed the new heterogeneous network to the homogeneous one. As Jaccard similarity is only available for the binary object, we just utilized dot product and cosine similarity to complete the transformation. Now we have obtained the repository network and to visualize the repository networks, firstly, we used Gephi [5] and Figure 20 is an example of the repository network transformed from Objective-C and C repository-stargazer network. From the figure, we can see that although we have already applied the function of preventing overlap, there is still a lot of vertices overlap and we also noticed that the red vertices are located close to each other, while the blue vertices are distributed dispersedly. In addition, after we applied the inverse document frequency, the two kinds of vertices are mixed together and it is impossible to find a borderline between them. The reason cause this is that the layout of the Gephi graph is one dimensional. To ameliorate this problem, we use Largevis[35] to visualize the networks. Largevis takes a weighted graph as input and utilizes node embedding technology and represents each vertex using a vector. Then it reduces vector length to two dimensions to implement network visualization.

As the weighted repository network could be represented as a square matrix and each repository could be represented as a vector, whose element is the distance between this repository to the others, reducing the vector length to two dimensions by Largevis, we get the coordinates of each repository. Then we plot the repository based



(a) Dot product (b) Cosine (c) Jaccard

(d) Dot product & IDF (e) Cosine & IDF

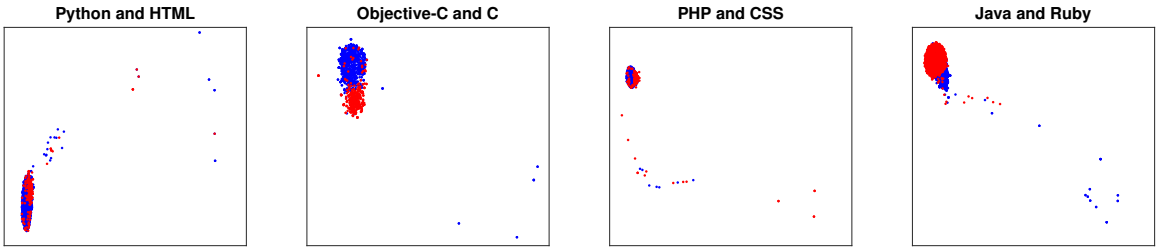
FIGURE 20: Gephi visualization of Objective-C and C repository-stargazer networks

on the coordinates by Matlab and Figure 21 and Figure 22 display the distribution of repositories of repository-contributor subnetworks and repository-stargazer subnetworks that transformed by different weighting schemes, respectively. From Figure 21, we found that for all subnetworks of repository-contributor network, no matter which weighting schemes we selected or whether applied the inverse document frequency or not, after the transformation, most of the repositories are distributed together. But there are also some repositories are located away from the principal part and after checking these repositories, we found that there is no connection between these repositories and the principal part and these repositories are usually only connected with one repository, which is also far away from the main part. And for the main part, in most cases, we can clearly distinguish two kinds of repositories as there is a clear borderline between two communities. While, there are some special cases that two categories are distributed together so that it is difficult to cluster, such as the Python and HTML network transformed by the combination of cosine similarity and inverse document frequency.

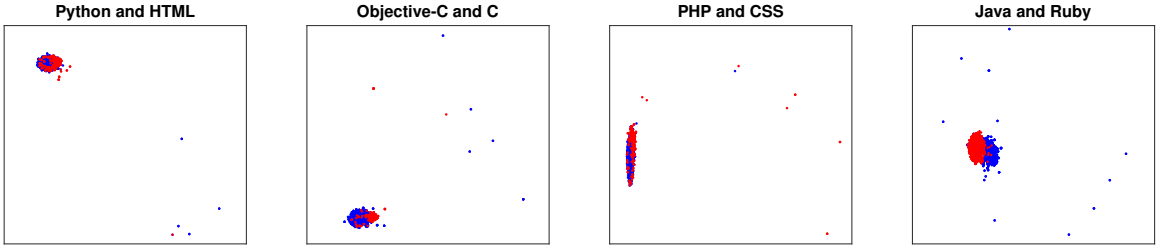
Unlike the repository-contributor subnetworks, for all weighting schemes, the repositories of all repository subnetworks, obtained by the subnetworks extracted from repository-stargazer network, are clearly distributed into two regions and there is only a few number of repositories are distributed into the incorrect communities. In addition, we found that after applying the inverse document frequency, the distribution of repositories is more sparsely. Therefore, comparing the distribution of two kinds of subnetworks, we could get better clustering results of repository subnetworks transformed by the repository-stargazer network than those transformed by the repository-contributor subnetworks.

4.2 SubNetwork Clustering Results

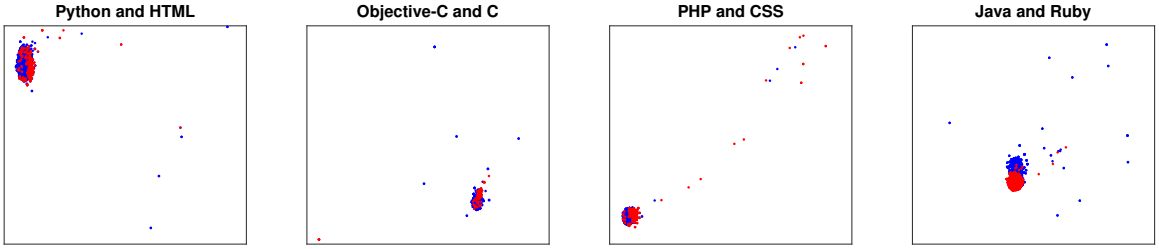
Then we clustered these repository networks by two clustering algorithms, greedy modularity maximization optimization algorithm and spectral modularity maximization optimization algorithm. Based on the clustering results, we reordered the repositories of each repository network and plotted their relations for different weighting



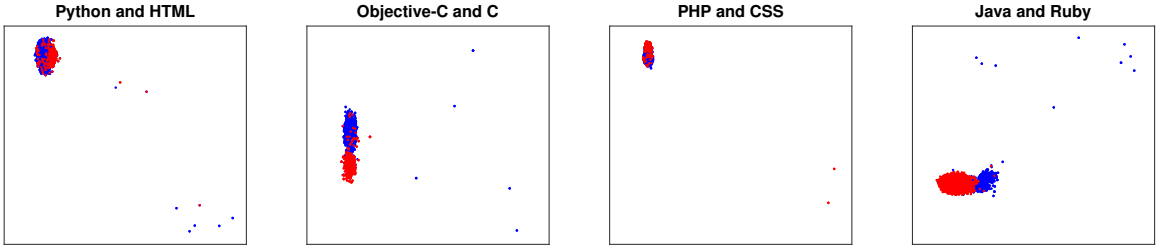
(a) Dot product



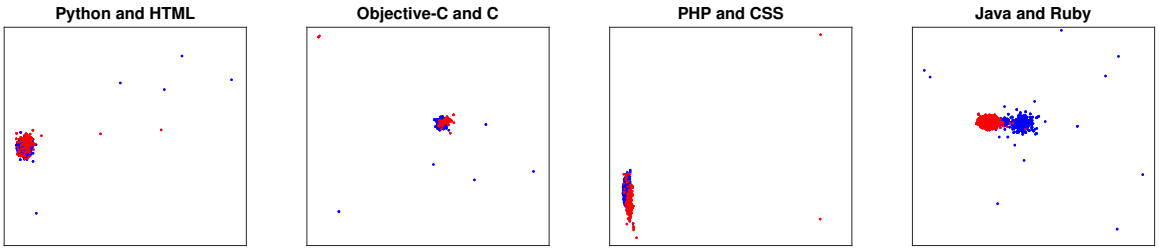
(b) Cosine



(c) Jaccard

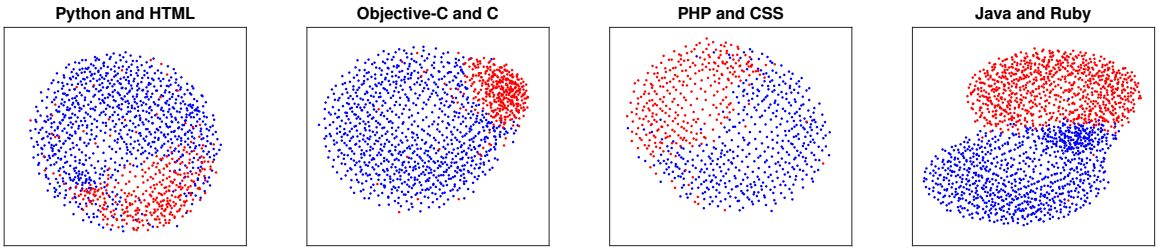


(d) Dot product + idf

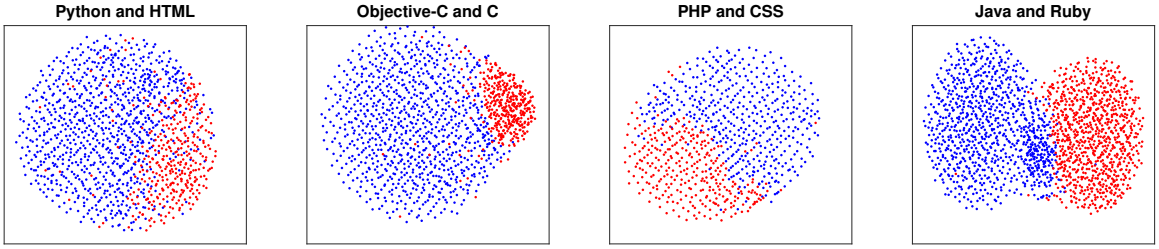


(e) Cosine + idf

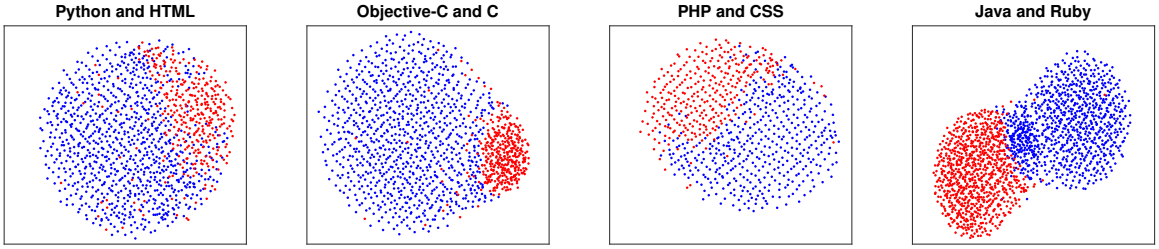
FIGURE 21: Repository distribution of subnetworks transformed from repository-contributor networks



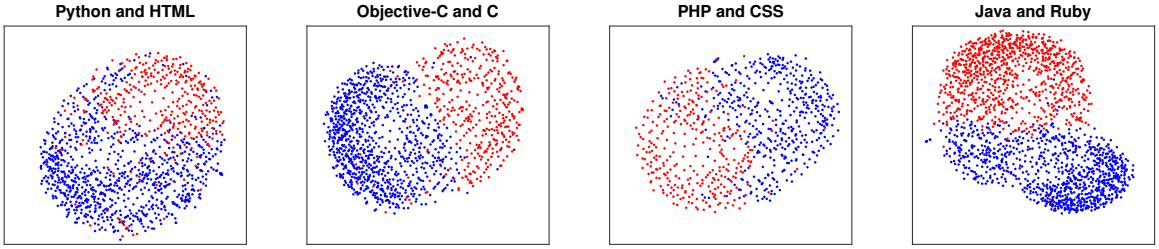
(a) Dot product



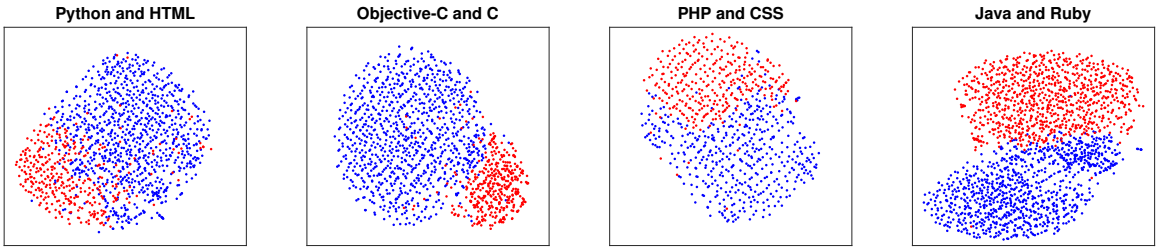
(b) Cosine



(c) Jaccard



(d) Dot product + idf

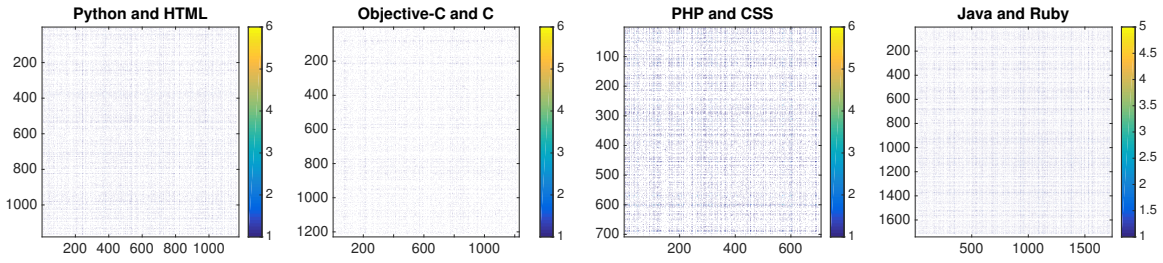


(e) Cosine + idf

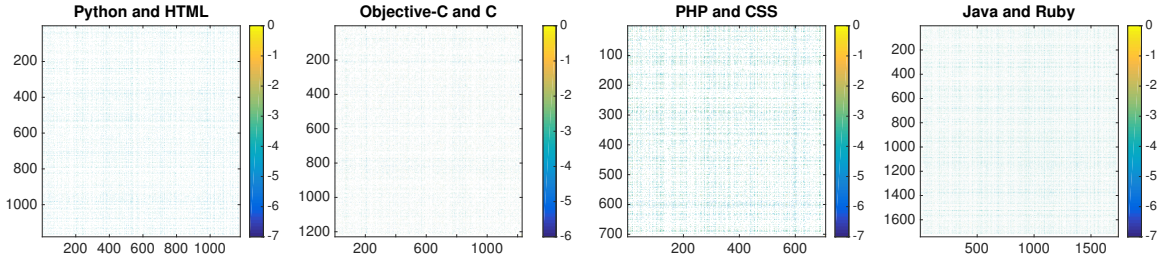
FIGURE 22: Repository distribution of subnetworks transformed from repository-stargazer subnetworks

schemes. Figure 23 and Figure 24 separately display the clustering results of the repository network, obtained from the repository-contributor network, for greedy clustering algorithm and spectral clustering algorithm. For the greedy algorithm, the figures of the same dataset are almost same. While, for the spectral algorithm, most of these figures also do not provide useful information and it is difficult to distinguish two communities from the figures. Table 9 displays the value of modularity and the evaluation measure values of community structures detected by the two clustering algorithms on these subnetworks. From the table, we noticed a strange phenomenon that no matter which weighting schemes we used to transform the repository-contributor network, the evaluation measures of the greedy algorithm is same for all four datasets. The result also explains why in Figure 23 all the figures of the same dataset are same. However, as the different weighting schemes we used, the modularity is different. We can see that without inverse document frequency, Jaccard similarity obtains the highest modularity, which is followed by cosine similarity and the value of dot product gets the lowest one. When we applied the inverse document frequency, the modularity of both dot product and cosine similarity increase and cosine similarity receives the highest modularity from all the weighting schemes. On the other hand, for the spectral algorithm, the combination of cosine similarity and inverse document frequency gets the highest value of both F-measure and rand index on Python and HTML dataset and PHP and CSS dataset. It also obtains higher F-measure(0.7510) than other weighing schemes on Objective-C and C dataset, but dot product receives the highest rand index(0.6323). For Java and Ruby, cosine similarity performs best for both evaluation measures. In addition, we found cosine similarity and Jaccard similarity gets higher modularity than dot product. Unlike the result of greedy algorithm, the inverse document frequency decreases the modularity for both dot product and cosine similarity.

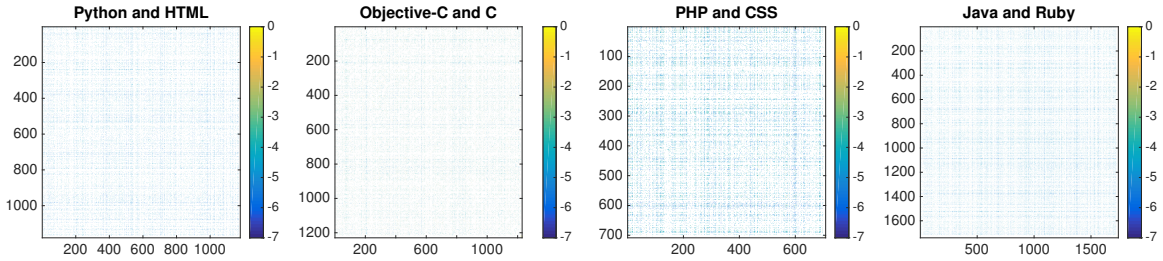
Then we clustered repository networks, obtained from the repository-stargazer network. Figure 25 and Figure 26 separately display the clustering results for both clustering algorithms. Unlike the network transformed from the repository-contributor network, this time, we can clearly differentiate two categories of repositories from all



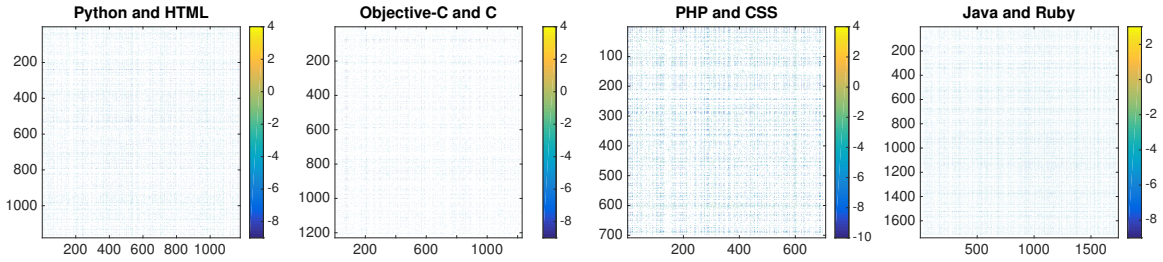
(a) Dot product



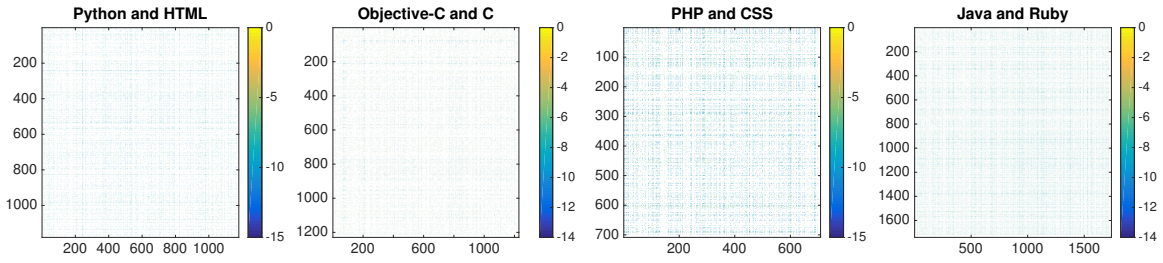
(b) Cosine



(c) Jaccard

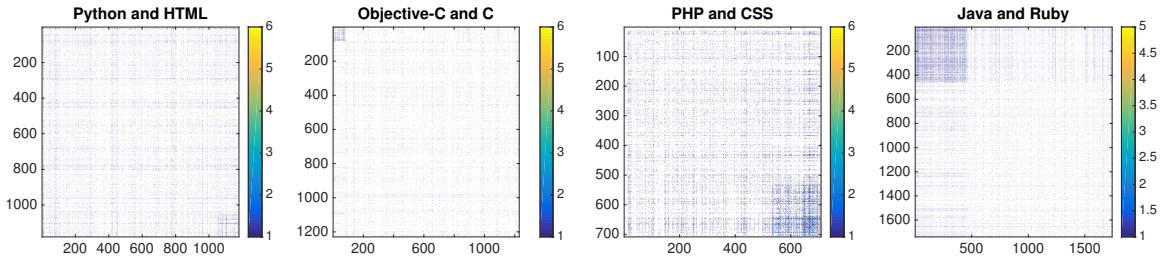


(d) Dot product + idf

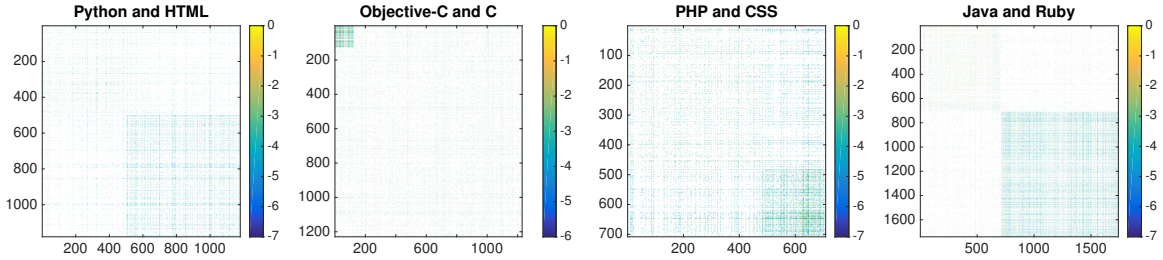


(e) Cosine + idf

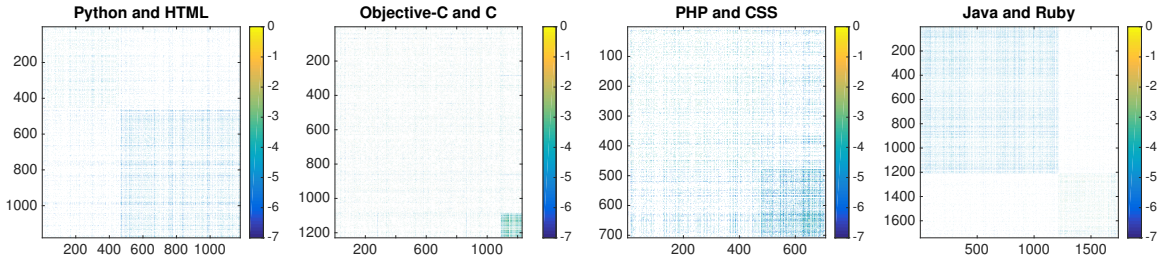
FIGURE 23: Clustering result by greedy algorithm of subnetworks transformed from repository-contributor subnetworks



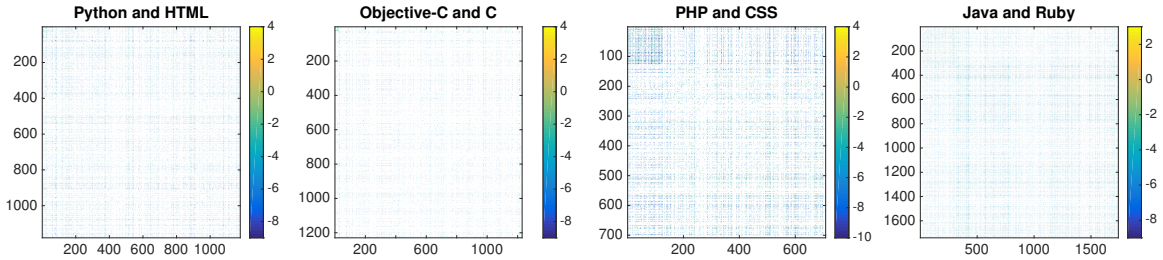
(a) Dot product



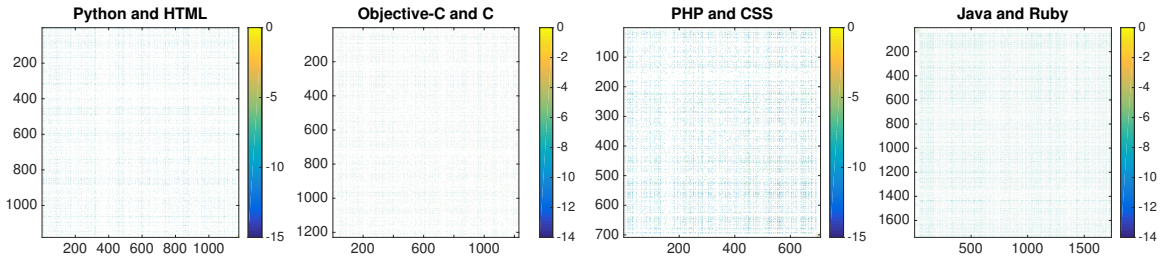
(b) Cosine



(c) Jaccard



(d) Dot product + idf



(e) Cosine + idf

FIGURE 24: Clustering result by spectral algorithm of subnetworks transformed from repository-contributor subnetworks

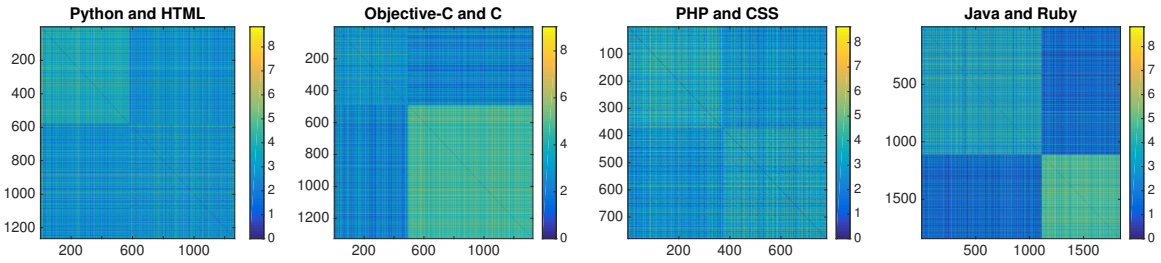
Dataset	Transform approach	Greedy Algorithm			Spectral Algorithm		
		M	F	RI	M	F	RI
Python and HTML	Dot product	0.0002	0.7598	0.6042	0.0572	0.7245	0.5524
	Cosine	0.0049	0.7598	0.6042	0.2608	0.7278	0.5696
	Jaccard	0.0076	0.7598	0.6042	0.2750	0.7284	0.5715
	Dot product + idf	0.0012	0.7598	0.6042	0.0765	0.7396	0.5715
	Cosine + idf	0.0321	0.7598	0.6042	0.0371	**0.7584**	**0.6003**
Objective-C and C	Dot product	0.0007	0.7479	0.5827	0.1787	0.7436	**0.6323**
	Cosine	0.0127	0.7479	0.5827	0.2859	0.7071	0.5280
	Jaccard	0.0249	0.7479	0.5827	0.2890	0.7061	0.5261
	Dot product + idf	0.0017	0.7479	0.5827	0.1501	0.7490	0.6099
	Cosine + idf	0.0742	0.7479	0.5827	0.0737	**0.7510**	0.5930
PHP and CSS	Dot product	0.0001	0.7247	0.5361	0.2311	0.6381	0.5034
	Cosine	0.0022	0.7247	0.5361	0.2345	0.6088	0.5118
	Jaccard	0.0022	0.7247	0.5361	0.2388	0.607	0.5123
	Dot product + idf	0.0007	0.7247	0.5361	0.1765	0.6641	0.4993
	Cosine + idf	0.0181	0.7247	0.5361	0.0514	**0.7209**	**0.5417**
Java and Ruby	Dot product	0.0002	0.6891	0.5000	0.1601	0.7503	0.6158
	Cosine	0.0051	0.6891	0.5000	0.3557	**0.9304**	**0.8329**
	Jaccard	0.0089	0.6891	0.5000	0.3125	0.8074	0.6784
	Dot product + idf	0.0013	0.6891	0.5000	0.1279	0.6410	0.4997
	Cosine + idf	0.0329	0.6891	0.5000	0.0830	0.6838	0.5016

TABLE 9: Evaluation result of subnetworks transformed from repository-contributor subnetworks, M: Modularity, F: F-measure, RI:Rand Index

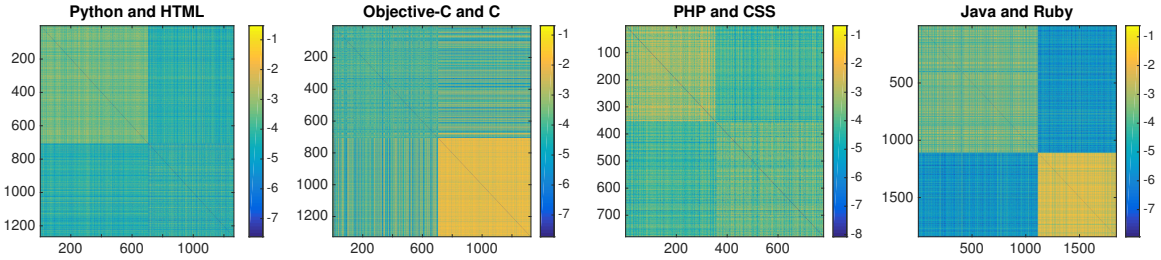
the figures, no matter which the weighting schemes or the clustering algorithms we applied, although in some figures there is only one distinct community. We also noticed that in most of the figures, the connection between the repositories in the same community is tighter than the repository in different communities. In addition, just considering the number of repositories of each community, the result of dot product is more like the ground truth than the other two weighting schemes. If we apply the inverse document frequency, both dot product and cosine similarity perform better than the original ones. Table 10 displays the value of modularity and the evaluation measure values of community structures detected by the two clustering algorithms on these subnetworks. For the modularity, we found that the inverse document frequency increases the value of both dot product and cosine similarity for both clustering algorithms. In addition, in PHP and CSS dataset and Java and Ruby dataset, cosine similarity and Jaccard similarity performs better and dot product. But in Objective-C and C dataset, dot product receives the highest value of modularity for both algorithms. In Python and HTML dataset, dot product and Jaccard similarity separately obtain higher value than the other two weighting schemes for greedy algorithm and spectral algorithm. When we cluster the subnetworks by the greedy algorithm, the combination of dot product and inverse document frequency performs better than the other weighting schemes and it gets the highest value of both evaluation measures on Python and HTML dataset, Objective-C and C dataset and Java and Ruby dataset. While on the PHP and CSS dataset, dot product obtains the best result value of F-measures and rand index, separately 0.8714 and 0.7740. However, for the spectral algorithm, the combination of dot product and inverse document gets the highest value for both evaluation measures on all four datasets.

4.3 Distribution of Weighting Schemes

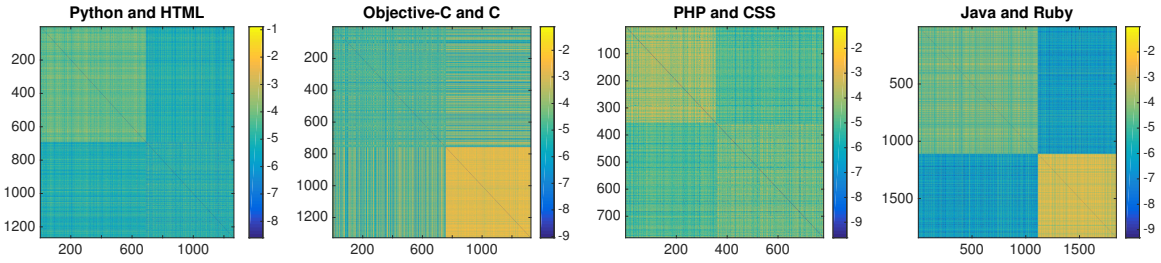
From the experiments, we fount that Dot product is best weighting schemes in most case and inverse document frequency greatly improve the performance of both dot product and cosine similarity. To explain this phenomenon, we firstly plotted the distribution of all weighting schemes. Figure 27 and Figure 28 separately plot the



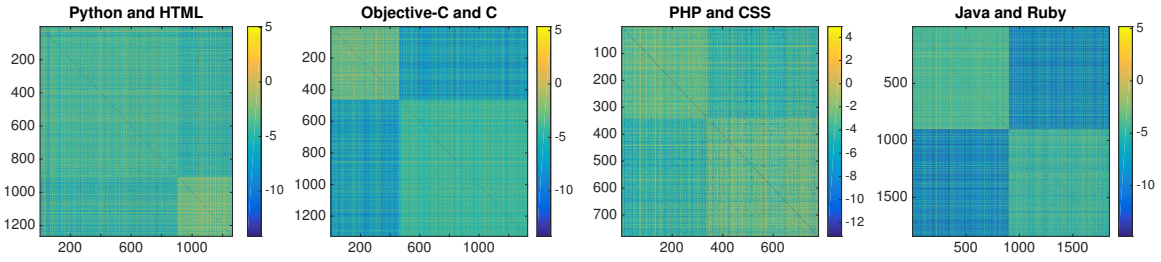
(a) Dot product



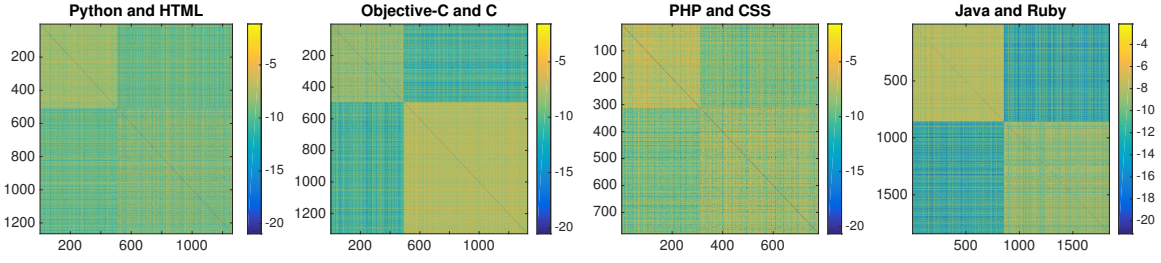
(b) Cosine



(c) Jaccard

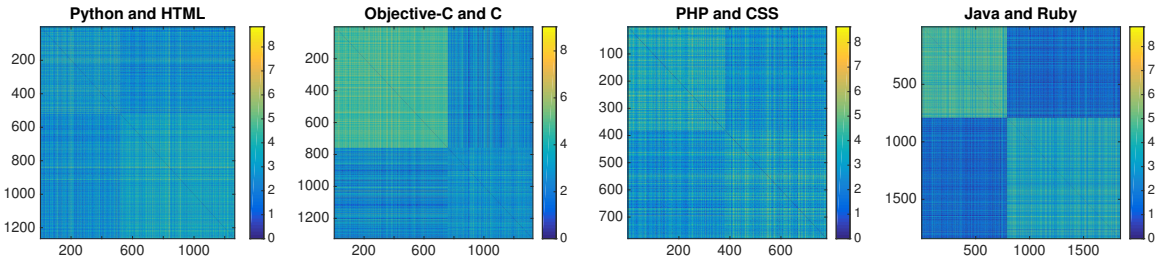


(d) Dot product + idf

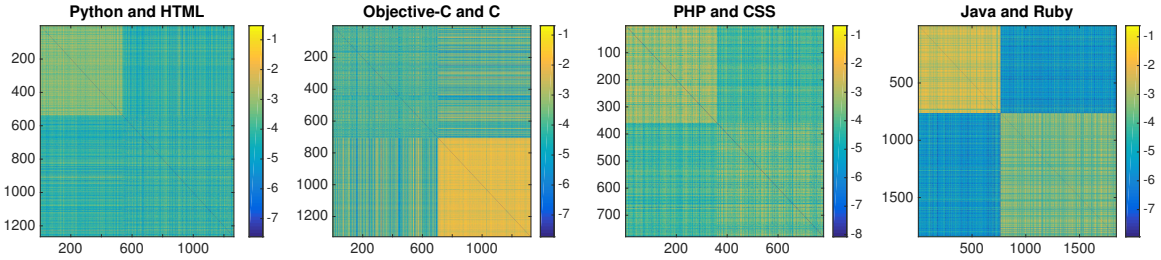


(e) Cosine + idf

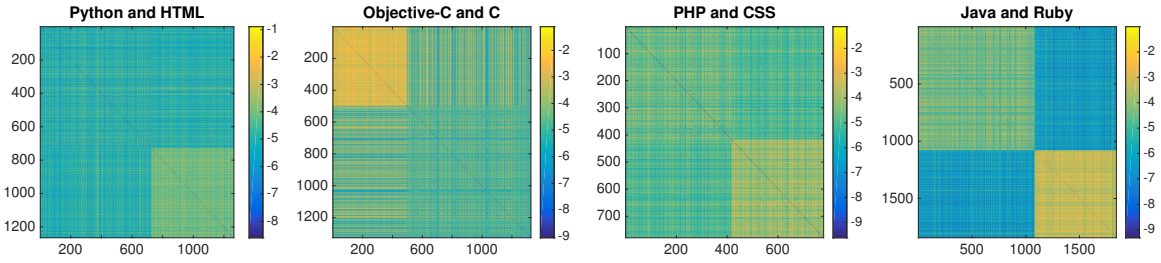
FIGURE 25: Clustering result by greedy algorithm of subnetworks transformed from repository-stargazer subnetworks



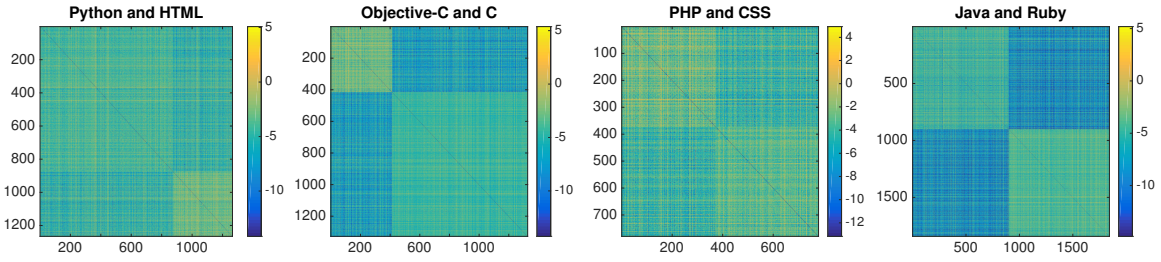
(a) Dot product



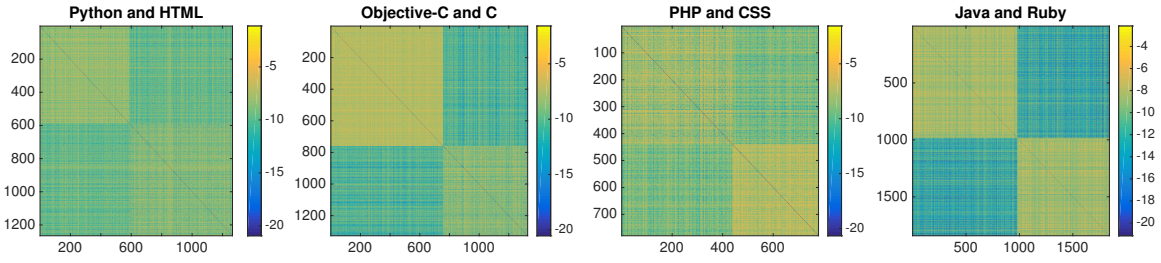
(b) Cosine



(c) Jaccard



(d) Dot product + idf



(e) Cosine + idf

FIGURE 26: Clustering result by spectral algorithm of subnetworks transformed from repository-stargazer subnetworks

Dataset	Transform approach	Greedy Algorithm			Spectral Algorithm		
		M	F	RI	M	F	RI
Python and HTML	Dot product	0.1185	0.7245	0.5884	0.1169	0.8278	0.7043
	Cosine	0.0969	0.7809	0.6455	0.1203	0.6983	0.5678
	Jaccard	0.1055	0.7902	0.6559	0.1229	0.6943	0.5649
	Dot product + idf	0.2020	**0.8743**	**0.7800**	0.2029	**0.8658**	**0.7649**
	Cosine + idf	0.2215	0.6681	0.5474	0.2198	0.7239	0.5878
Objective-C and C	Dot product	0.0974	0.9040	0.8219	0.0965	0.8623	0.7545
	Cosine	0.0940	0.7593	0.6238	0.0936	0.7585	0.623
	Jaccard	0.0937	0.7196	0.5871	0.0914	0.6699	0.5514
	Dot product + idf	0.3499	**0.9189**	**0.8478**	0.3394	**0.9432**	**0.8919**
	Cosine + idf	0.2369	0.9048	0.8231	0.2384	0.8645	0.7577
PHP and CSS	Dot product	0.1695	**0.8714**	**0.7740**	0.1690	0.8829	0.7914
	Cosine	0.1722	0.8508	0.7444	0.1721	0.8573	0.7535
	Jaccard	0.1766	0.8496	0.7426	0.1763	0.8521	0.7462
	Dot product + idf	0.2770	0.8339	0.7216	0.2333	**0.9045**	**0.8258**
	Cosine + idf	0.2261	0.8008	0.6808	0.2822	0.8274	0.7131
Java and Ruby	Dot product	0.3999	0.8677	0.7724	0.4000	0.9090	0.8350
	Cosine	0.4079	0.8694	0.7749	0.4073	0.8935	0.8106
	Jaccard	0.4108	0.8688	0.7740	0.4100	0.8868	0.8004
	Dot product + idf	0.3206	**0.9756**	**0.9523**	0.3166	**0.9582**	**0.9198**
	Cosine + idf	0.4037	0.9723	0.9461	0.4066	0.9391	0.8857

TABLE 10: Evaluation result of subnetworks transformed from repository-stargazer subnetworks, M: Modularity, F: F-measure, RI:Rand Index

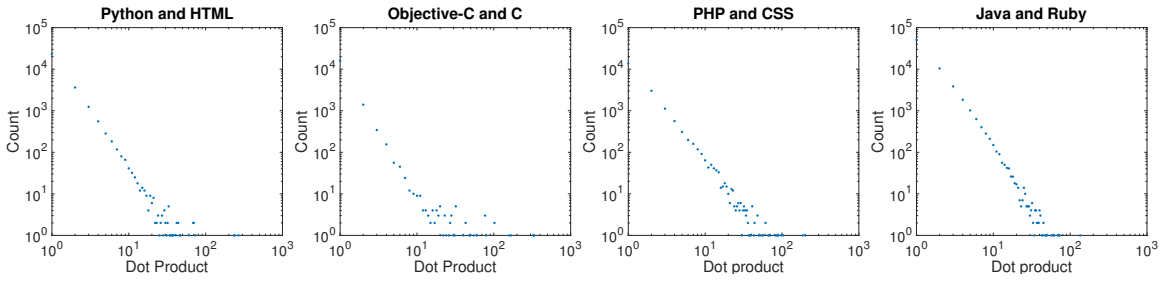
distribution of weighting schemes of repository subnetworks obtained by the transformation of repository-contributor subnetworks and repository-stargazer subnetworks. From Figure 27, we noticed that for each weighting schemes, the distribution of all four subnetworks is shown the same trend. For both dot product and the combination of cosine similarity and inverse document frequency, the distribution follows a power law distribution. Without applying the inverse document frequency, the distribution of cosine similarity increases at first and reach the peak around 0.01. Then it begins to decrease and follows a power law distribution. However, the distribution of Jaccard similarity is strange. At the beginning, it climbs to the top when the value of Jaccard similarity is around 0.005 and then begins to drop. But during the process of reducing,with the increase of Jaccard similarity value, there is a gap for the count. In addition, this phenomenon is also shown in the distribution of the combination of dot product and inverse document frequency, although it follows a power law distribution. Comparing the three weighting schemes, we found that for all the four datasets, the count of dot product, which is equal to 1, is the largest one, but the large count of cosine similarity and Jaccard similarity obtains the largest count at the medium value. If we apply the inverse document frequency, we noticed although there is a large number weight gets small value, there is also a lot of weight get higher values. However, the distribution of cosine similarity follows a power law distribution. From the result we found that the combination of cosine similarity and inverse document frequency achieves the best performance, so I think that if the distribution follows a power law distribution, the network may be clustered more like the original one.

For the repository subnetworks transformed from the repository-stargazer subnetworks, the distribution of dot product, the combination of dot product and inverse document frequency and the combination of cosine similarity and inverse document frequency follows a power law distribution. While for the other two weighting schemes, except the increase at the start, they also follow a power law distribution. Besides we also noticed that the distribution of Objective-C and C dataset transformed by cosine similarity and Jaccard similarity has a phase that the count of value remains constant. Comparing the distribution of dot product, cosine similarity, and Jaccard similarity,

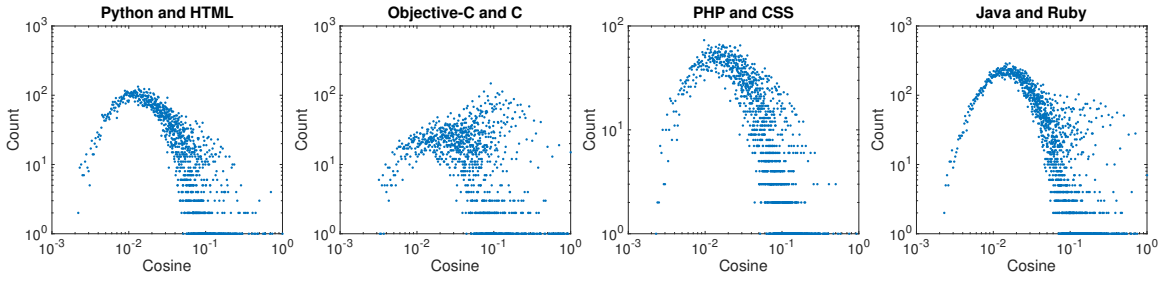
we found that there is a lot of weight gets small values of dot product, but the number of small weight of cosine similarity and Jaccard similarity is far less than dot product. Most of the weight of cosine similarity and Jaccard similarity gets medium value. This may cause the performance of cosine similarity and Jaccard similarity worse than dot product. However, when we apply the inverse document frequency, cosine similarity shows the same distribution as dot product, and this explains why the inverse document frequency significantly improves the value of evaluation measures. In addition, we found that after the inverse document frequency, the distribution of dot product is tighter than the original one and it also performs better.

4.4 Visualization of Clustering Results

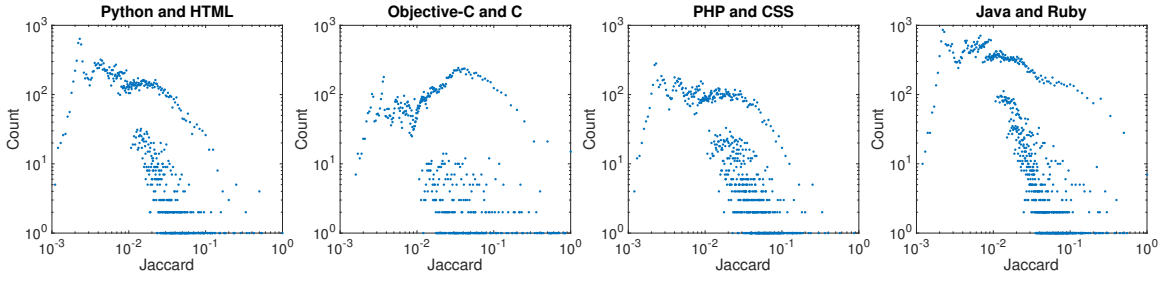
To find out the reason of the incorrect clustering, we labeled those repositories, which are clustered into the wrong communities, on the distribution of repositories. Because the clustering result of the repository network transformed from the repository-contributor is not good, we just plot those repository network obtained from the repository-contributor network. Figure 29 and Figure 30 separately display the clustering result of greedy modularity maximization optimization algorithm and spectral modularity optimization algorithm on all subnetworks transformed by different weighting schemes. In these figures, each colour represents a kind of repositories of the original label. The point means the repository is clustered into the correct community and the cross means the repository is clustered into the wrong community. We noticed that except those repositories are distributed into the other community are clustered incorrectly, most of the repositories clustered mistakenly is located on the border of two communities. In addition, we can clearly found that before applying the process of inverse document frequency, when we using dot product to transform the network, there are fewer repositories are clustered into the wrong communities. We also found that the inverse document frequency also decreases the rate of incorrect clustering, which is just as we have shown in 25 and Figure 26. Then we separately selected a repository, which is incorrectly clustered for all the five different weighting scheme subnetworks, from four datasets and the detail of the repositories are shown



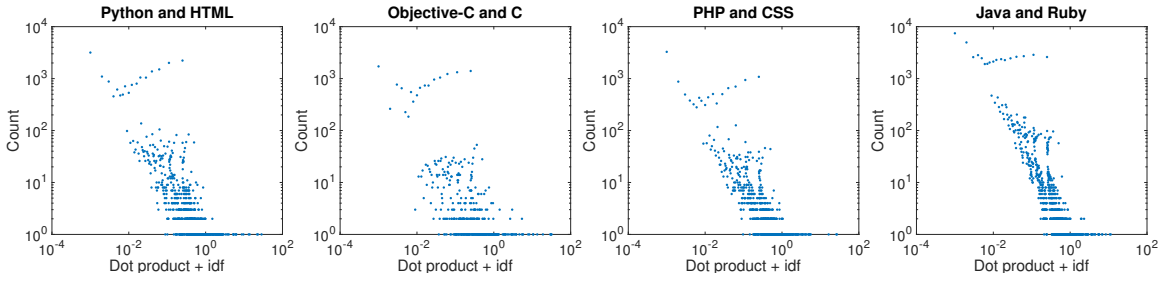
(a) Dot product



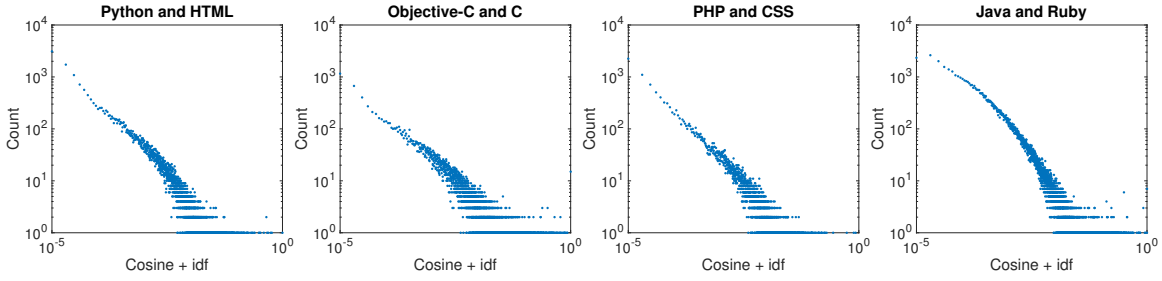
(b) Cosine



(c) Jaccard

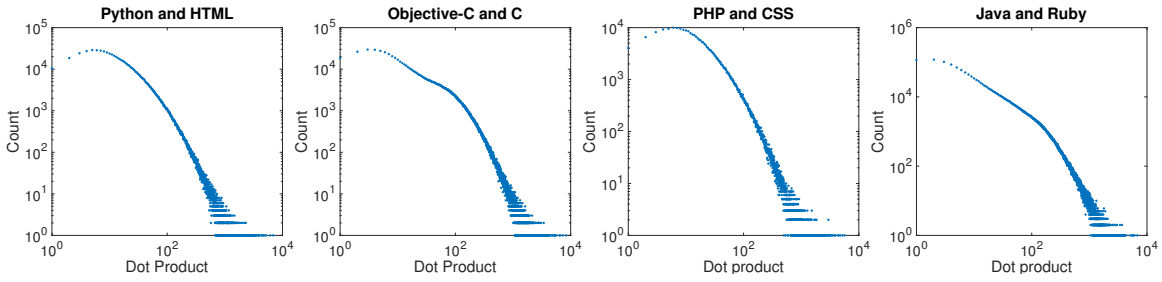


(d) Dot product + idf

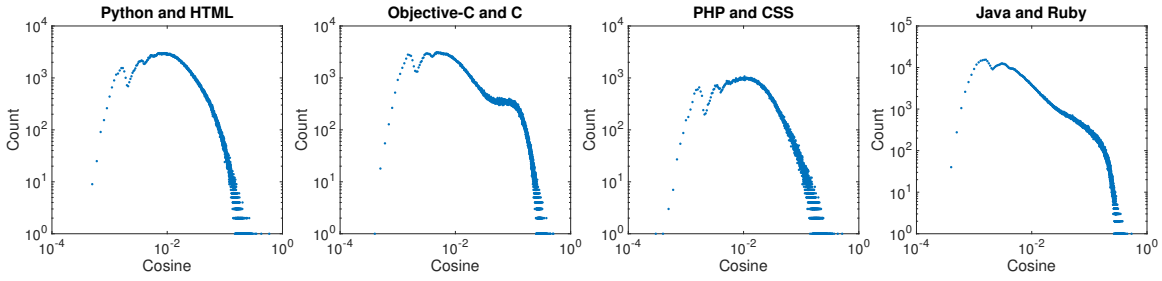


(e) Cosine + idf

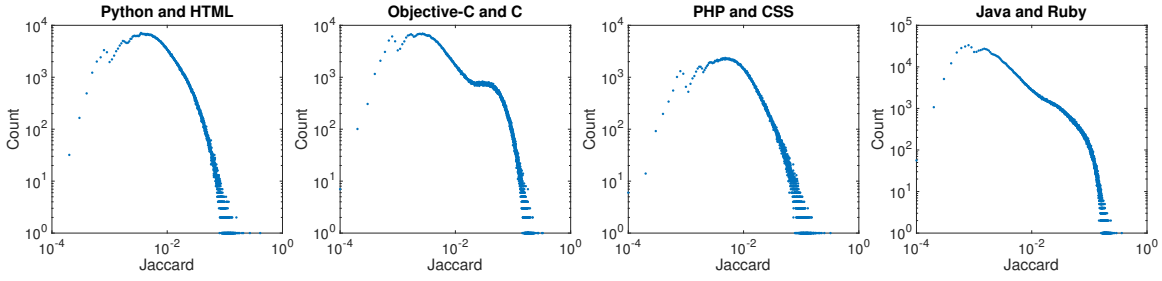
FIGURE 27: Weighting schemes distribution of subnetworks transformed from repository-contributor subnetworks



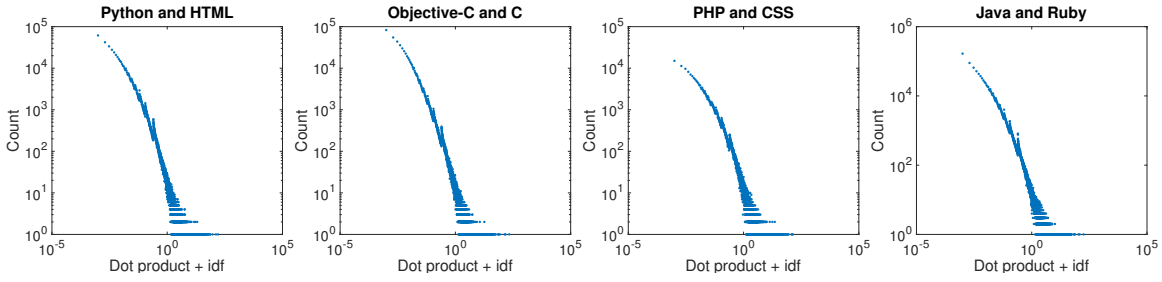
(a) Dot product



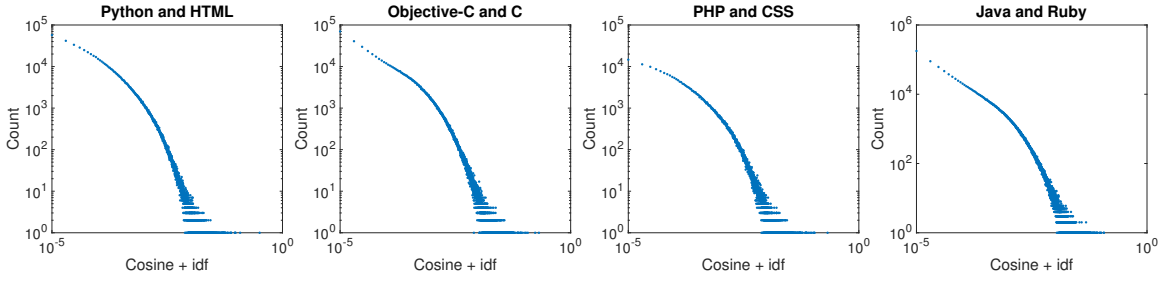
(b) Cosine



(c) Jaccard



(d) Dot product + idf



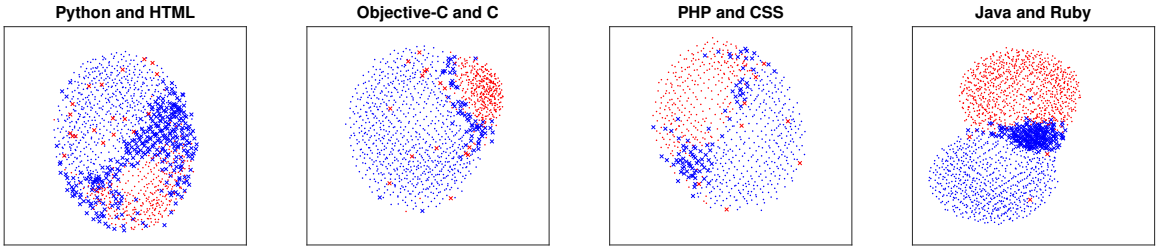
(e) Cosine + idf

FIGURE 28: Weighting schemes distribution of subnetworks transformed from repository-stargazer subnetworks

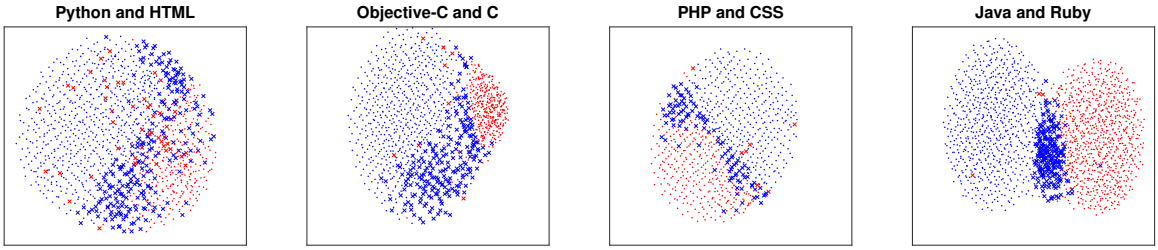
in Table 11. We can see that all of these repositories are written by at least three different programming languages and both of the languages we selected to labeled the dataset are included these programming languages. And these two languages are usually widely used in these repositories. So the composite application of programming languages may be the main reason cause the incorrect clustering.

5 Summary

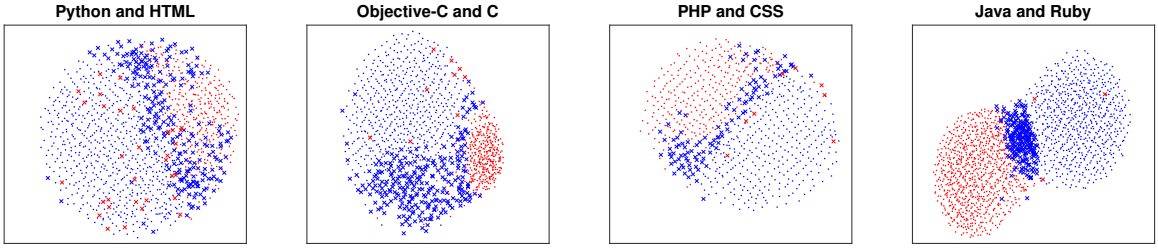
This section discussed the heterogeneous-transformed homogeneous clustering method for the homogeneous network clustering. We selected three different categories weight, dot product, cosine similarity and Jaccard similarity to transform the heterogeneous network to a homogeneous one.Then we use modularity optimization algorithm cluster the homogeneous network and evaluated the result by F-measure and rand index. Table 12 lists the weighting schemes when we get the best results on different datasets for both algorithms, and we found that for the repository-stargazer network, the combination of dot product and inverse document frequency perform better than other weight schemes for both clustering algorithms. As the sparsity of repository-contributory, the clustering result is not satisfied. But we also found that we usually get the same clustering result of the greedy modularity maximization optimization algorithm, no matter which weighting schemes we select or whether apply the inverse document frequency. However, if we cluster the homogeneous by the spectral algorithm, the combination of cosine similarity and inverse document is the best choice. Therefore, for GitHub dataset, repository-stargazer network provides more information and dot product-idf is the best weighting schemes to describe the relation between repositories.



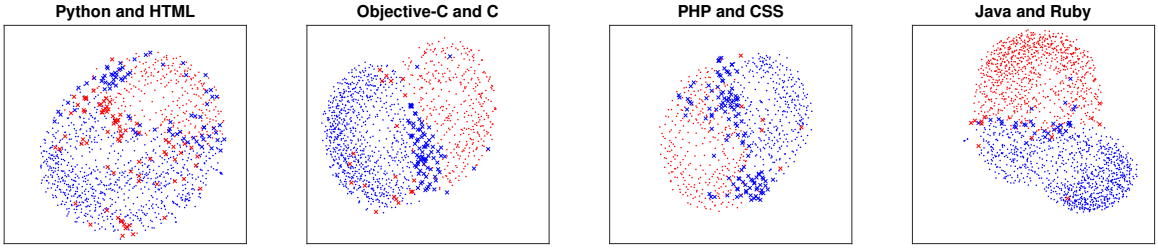
(a) Dot product



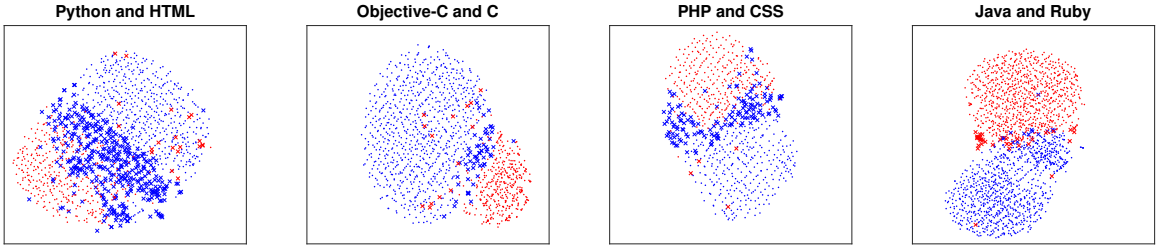
(b) Cosine



(c) Jaccard

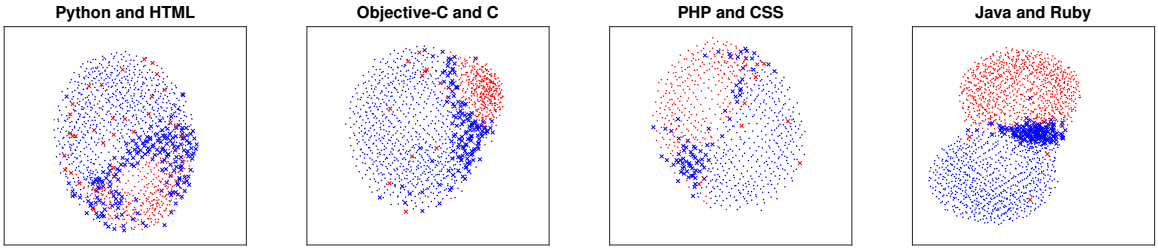


(d) Dot product + idf

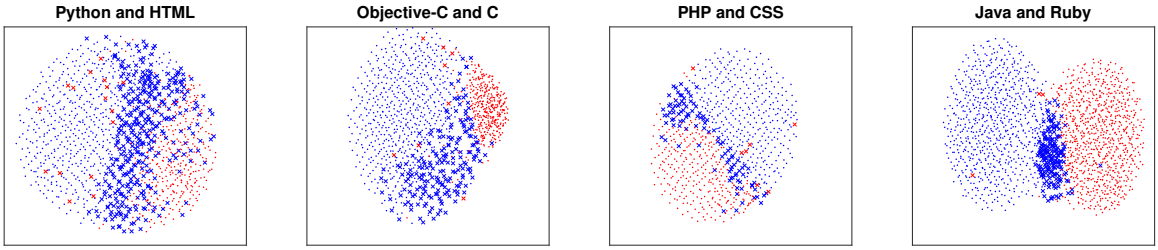


(e) Cosine + idf

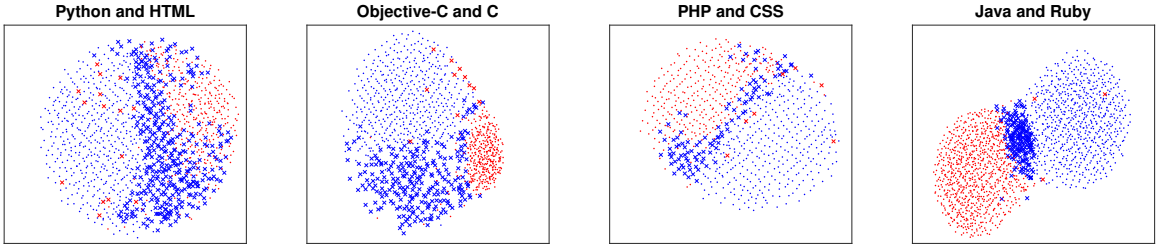
FIGURE 29: Repository distribution labeled by greedy algorithm clustering results of subnetworks transformed from repository-stargazer subnetworks, point means repository is clustered into correct communities and cross means repository is clustered into wrong communities



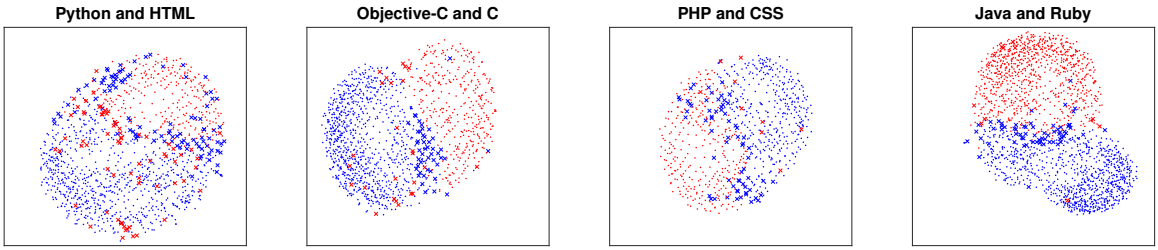
(a) Dot product



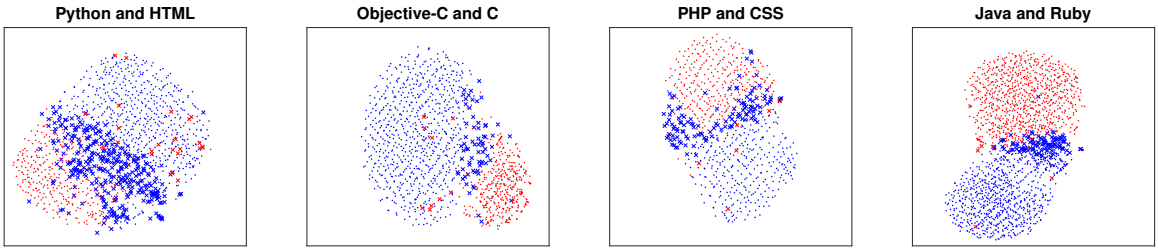
(b) Cosine



(c) Jaccard



(d) Dot product + idf



(e) Cosine + idf

FIGURE 30: Repository distribution labeled by spectral algorithm clustering results of subnetworks transformed from repository-stargazer subnetworks, point means repository is clustered into correct communities and cross means repository is clustered into wrong communities

Repository ID	Dataset	Labeled Language	Actual Language
9089664	PHP and CSS	PHP	PHP 273765 JavaScript 235403 CSS 145940
33945	Java and Ruby	Java	Java 716739 Ruby 611648 C 189161 HTML 96249 Yacc 7381 Shell 3964 XSLT 1868
11766695	Objective-C and C	C	C 1359072 Objective-C 212092 C++ 98486 Ruby 1514
27534934	Python and HTML	Python	Python 779863 HTML 62123 CSS 5766 JavaScript 2447 Makefile 399

TABLE 11: Example of repositories that are clustered into the wrong communities by two clustering algorithms for all weighting schemes

Dataset	Contributor		Stargazer	
	Greedy	Spectral	Greedy	Spectral
Python and HTML	all	C + idf	D+idf	D +idf
Objective-C and C	all	C+idf	D+idf	D+idf
PHP and CSS	all	C+idf	D	D+idf
Java and Ruby	all	C	D+idf	D+idf

TABLE 12: Weighting schemes for best result

CHAPTER VI

Communities in Repositories

In previous sections, we have found that dot product-idf is the best weighting scheme to describe the relations between repositories. This section uses this weighting scheme to reveal the overall relationship between repositories and their communities.

1 Visualization of the Original Clusters

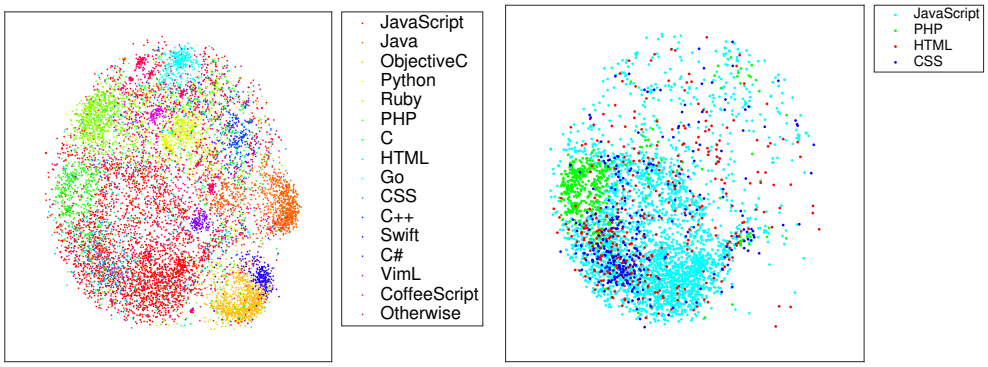
Based on the whole repository-stargazer network that has been collected, we first transform the heterogeneous network to homogeneous repository network using dot product-idf and each repository can be represented as a vector, whose element is the distance from this repository to the others. This vector is further reduced to two dimensions using LargeVis . All the repositories are plotted in fig 31(a). In this figure, repositories with popular languages(at least has over 100 repositories) are labeled by different colours, and the rest of repositories are belong to otherwise. To clearly observe the relationship between different communities, we subplot some figures only including a few number of programming languages repositories. The result is shown in Figure 31. From Figure 31(b), we can see that the distance between JavaScript with PHP is close and the repository of JavaScript, HTML and CSS and mixed together. Then we removed JavaScript repositories(Figure 31(c)), we found that the distance between HTML and CSS repository are tight, which means these two kinds of programming languages are usually used together. Then we plot the JavaScript and Java repositories(Figure 31(d)), and we can clearly observe that these two categories of repositories are distributed into different areas and there is just

a few repositories are close to the other kind of repositories. In Figure 31(e), we can see that the communities of Python, C, Go, and C++ are close to each other. While the repositories of C# are distributed in the same area, which is far away from the communities of the other four programming languages. At last, we plot the repositories with Objective-C, Ruby, and Swift. From Figure 31(f), it is clearly shown that Objective-C is close to Swift, and both languages are used for OS X and iOS operating systems. However, the communities of Ruby are located away from these communities in the contrary direction.

2 Clustering Results

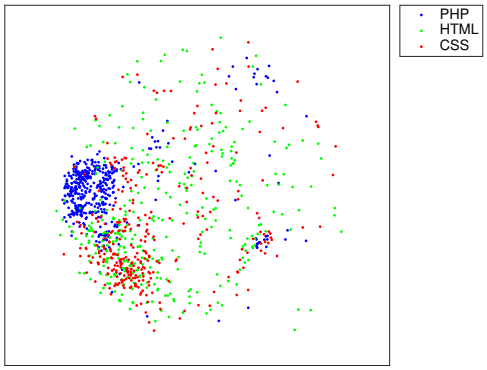
Then we applied greedy modularity optimization algorithm and spectral modularity optimization algorithm on the repository network. Figure 32 separately show the clustering results of these two algorithms. For greedy modularity optimization algorithm(Figure 32(a)), when we detect 7 communities, we get the highest modularity, 0.3050. At the same time, the value of F-measure and rand index is 0.4888 and 0.7932, respectively. Comparing with the original clusters, we can see that Ruby repositories belong to community 1. Community 2 includes Python, C, C++, and Go repositories and in reality, people who like to use C also prefer to utilize C++, Python, and Go. Objective-C and Swift form communities 3 and both programming languages are designed by Apple Inc. Community 4 is mainly composed by repositories from JavaScript, HTML, PHP, and CSS. All of these programming languages are used for web developing. Java and C# separately constitute community 5 and 6. Community 7 only includes 21 repositories, including 8 ActionScript repositories, 4 Haxe repositories,4 JavaScript repositories, 2 C++ repositories, 2 Shell repositories and 1 Dare repository.

Figure 32(b) show the clustering result of spectral modularity optimization algorithm. From the figure we can see there are totally four communities. At this time, the value of modularity, F-measure and rand index are separately 0.253, 0.3287and 0.7009. All of these value are lower than the result of greedy algorithm, which means

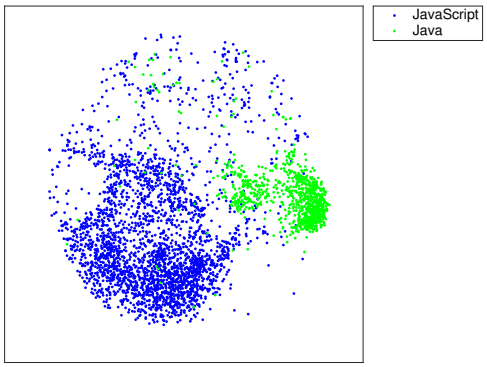


(a) Whole network visualization

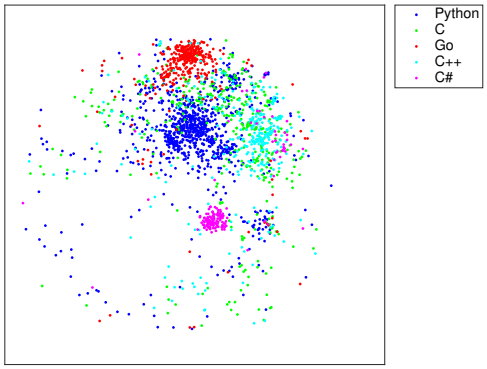
(b) JavaScript, PHP, HTML, CSS



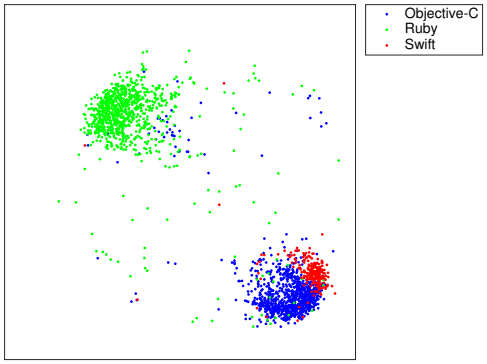
(c) PHP, HTML, CSS



(d) JavaScript, Java

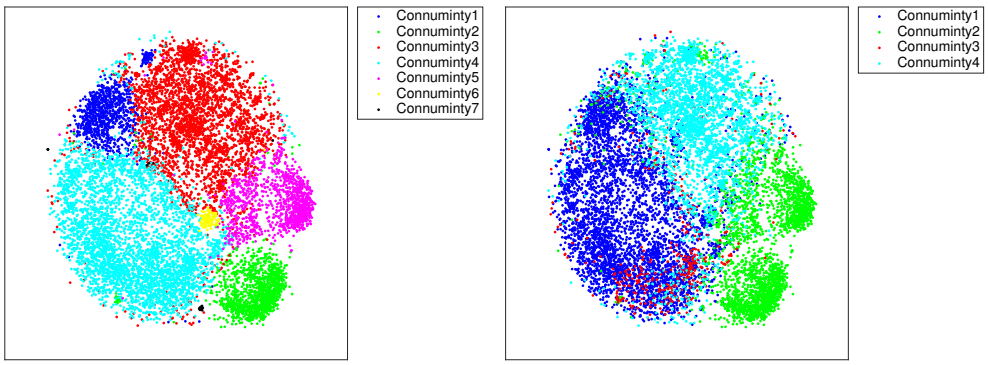


(e) Python, C, Go, C++, C#



(f) Objective-C, Ruby, Swift

FIGURE 31: Original communities visualization



(a) Greedy (b) Spectral

FIGURE 32: Clustering results visualization

the result of greedy algorithm is closer to the ground truth. The four communities we have detected separately includes 4497, 2743, 398 and 3027 repositories. Unlike the greedy algorithm, the spectral algorithm merges Ruby repositories into the community 1 of JavaScript, HTML, PHP, and CSS. Java, Objective-C, and Swift compose community 2. Community 3 is located between Community 1 and 2, and most of the repositories of this community are JavaScript repository. Community 4 is mainly made up of Python, C, C++ and Go repositories.

Comparing the results, we found the spectral algorithm detect fewer communities than greedy algorithm. Both clustering results present the relationship between communities of repositories. Each community is constructed by a serious of repositories with different programming languages. We can found that the programming languages belong to the same community are usually used together in practice. For instance, HTML, PHP, and CSS are used together to create web pages and web applications. Therefore, the clustering result shows the real reaction between programming languages in some degree.

CHAPTER VII

Relationship between Programming Languages

In this chapter, we investigate the relation between programming languages. We use three methods to cluster programming languages. One is based on the interaction between presuming languages and repositories and another is based on the relation between programming languages and users. In addition, we also reduced the matrix of programming languages-repositories to 2 dimensions and then detect the communities of programming languages.

1 Clustering using repositories

Now we can study the relationship between programming languages based on repository interactions. First, we need to find a vector representation for each language. We define the vector representation of language L is the sum of all the vectors of the repositories in language L, i.e.,

$$L = \sum_{R_i \in L} R_i, \tag{1}$$

where R_i is the vector representation of the i-th repository that is discussed in the last section. For example, Table 13(a) show a repository-repository matrix. Suppose repository R1, R2 are labeled by language L1, and R3, R4 belong to L2. Then we separately add the first , the second row, and the third, the fourth row together. We get the language-repository matrix and each language is represented as a vector.

	R1	R2	R3	R4
R1	0	5/16	1/16	5/16
R2	5/16	0	9/16	1/16
R3	1/16	9/16	0	5/16
R4	5/16	1/16	5/16	0

(a) Repository-repository matrix

	R1	R2	R3	R4
L1	5/16	5/16	10/16	6/16
L2	6/16	10/16	5/16	5/16

(b) Language-repository matrix

TABLE 13: Repository-repository-matrix transform toLanguage-repository Matrix

Based on the result of the last section, the most promising representations are dot product-idf. Hence, in the following discussions, we will focus on the weighting scheme only.

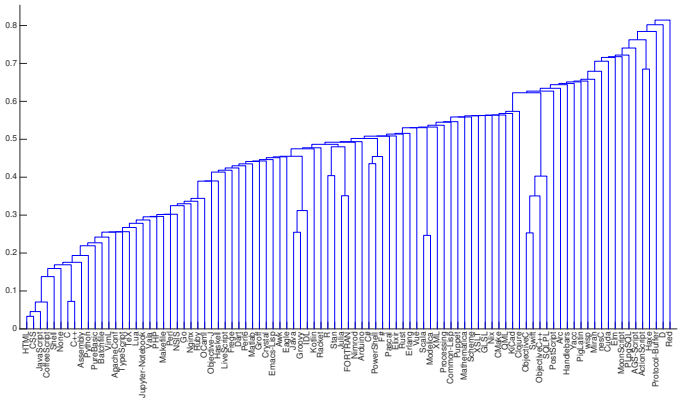
Once the language vectors are available, we can calculate their pairwise distances. We use cosine distance for the vectors. To reveal the hierarchical structure of programming languages, we apply HAC (hierarchical agglomerative clustering).

When running the HAC algorithms, the choice of the distance function for clusters is important. We tried several cluster distance functions, including single, complete, average, and weighted. Single calculates the shortest distance between two clusters. On the contrary, complete calculate the furthest distance. Average represents the unweighted average distance and weighted means the inner squared distance.

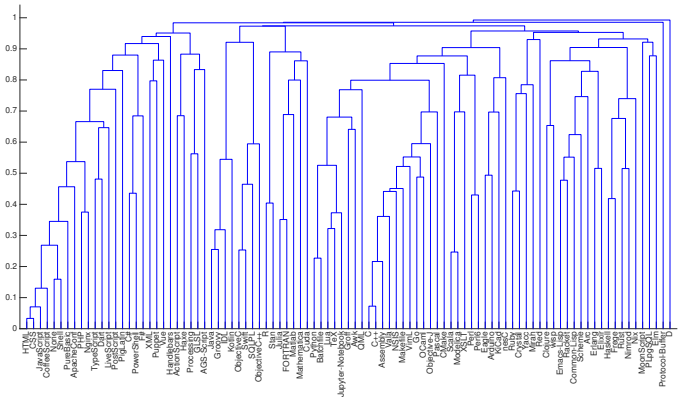
Figure 33 show the dendrograms generated from various distance functions. From the plots, we can see that mostly the clustering results are similar, but weighted is slightly better than the others. In the following, we discussed the details of the clustering result using weighted as cluster distance function.

From the dendrogram (Figure 34) and the heatmap (Figure 35) we can observe that

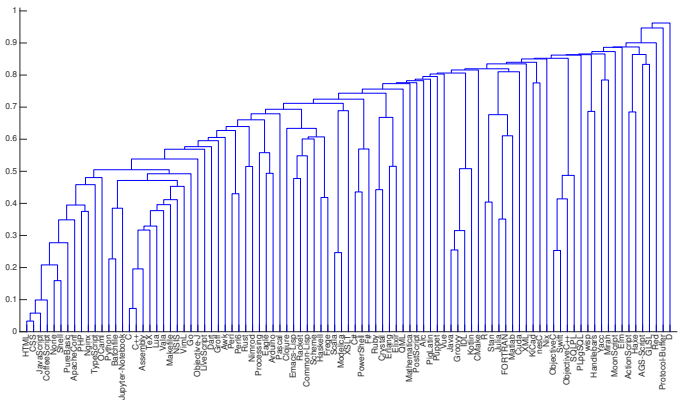
- For the dendrogram tree of dot product-idf(Figure34), there are mainly three clusters. The first one is from HTML to Puppet. In this cluster, HTML, CSS, JavaScript, and PHP are usually applied for web development. While Python, C, C++, and Go are used for system programming languages and python could also be used for web developing. Another community is the functional programming languages. From Scala to Red is the second clusters. In this cluster, most



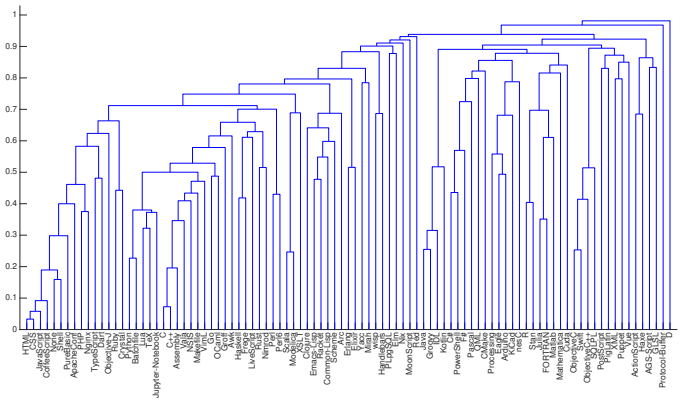
(a) single



(b) complete



(c) average



(d) weighted

FIGURE 33: Dendrogram Tree for different linkage

of the programming languages are not popular. And the rest programming languages constitute the last communities. This cluster includes Objective-C, Swift, and Objective-C++, which are designed by Apple Inc. for OS X and iOS operating system. And Java is also in this clusters with some other program languages, which is applied with Java or on the platform developed by Java, such as IDL and Groovy.

- Based on the order of the dendrogram tree, we separately draw the heatmap for both clustering schemes. In Figure 35, we can see the distance between HTML, CSS and JavaScript are close and all these programming languages are usually used together for web developing. We also found python is popular language and connect with a lot of other languages. The connection between C and C++ is tight because C++ is generated from C. Java is close to IDL , which is used to describe interface written by Java, and Groovy, an object-oriented programming language for Java platform. Besides, Objective-C, Swift, and Objective-C++ connect tightly with each other as all of them designed by the same company, Apple.

2 Clustering using users

During the process of network transformation, we lost some information which may affect the relation between programming languages. To solve this problem, we directly qualify the relation based on the repository-stargazer network by mutual information(MI). Since each repository is labeled by one programming languages, we can transfer the repository-stargazer network to the language-stargazer matrix. If the user stars the repository labeled by the programming language, the value of the row of the programming language and the column of the user is 1, otherwise is 0. For example, Table 14(a) show a repository-user matrix. Suppose repository R1, R2 are labeled by language L1, and R3, R4 belong to L2. We can see that both R1 and R2 are not connected with U6, so in Table 14(b), for L1, the value of column U6 is 0.

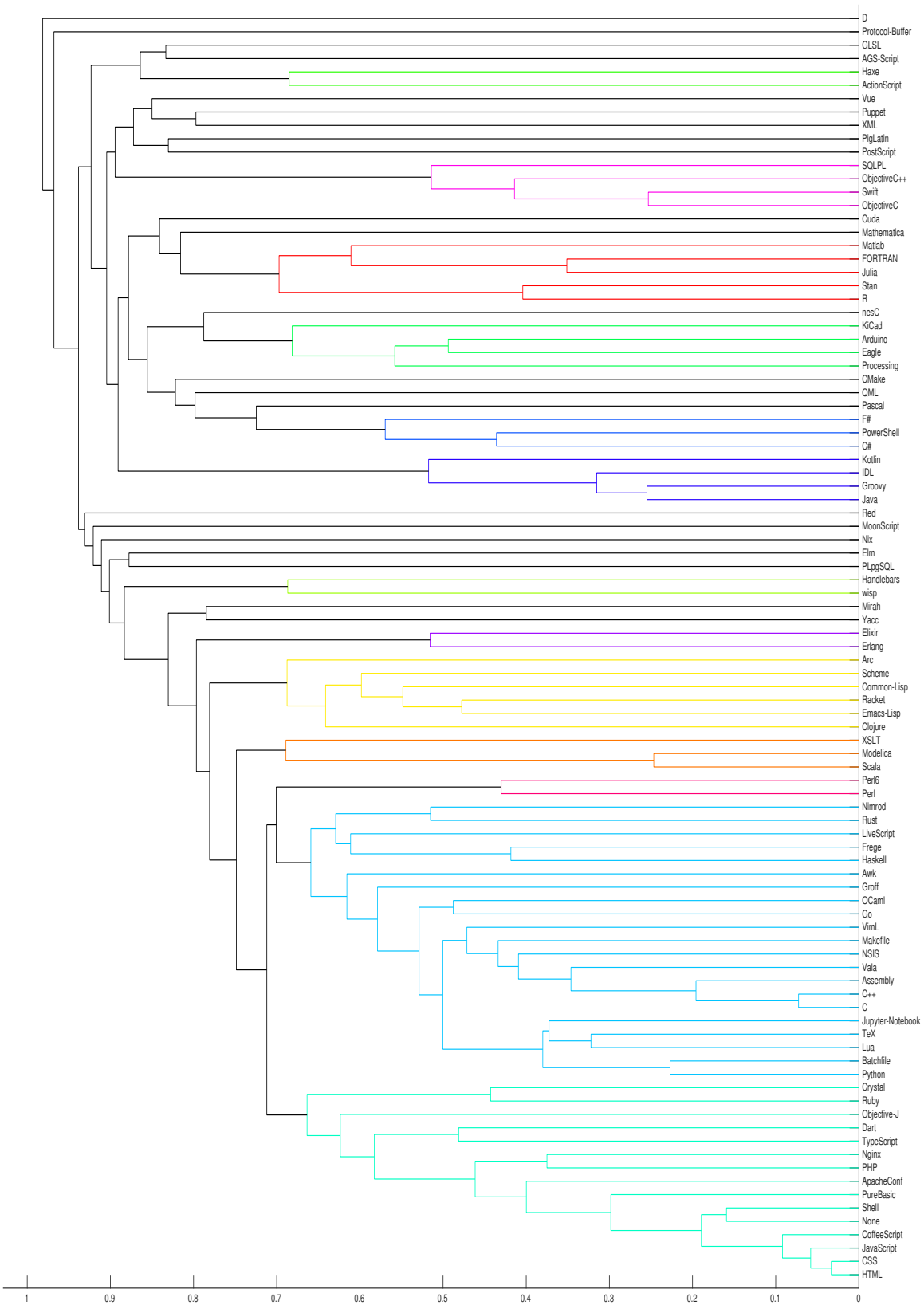


FIGURE 34: Dendrogram tree of all programming languages using cosine as repository distance and weighted as cluster distance

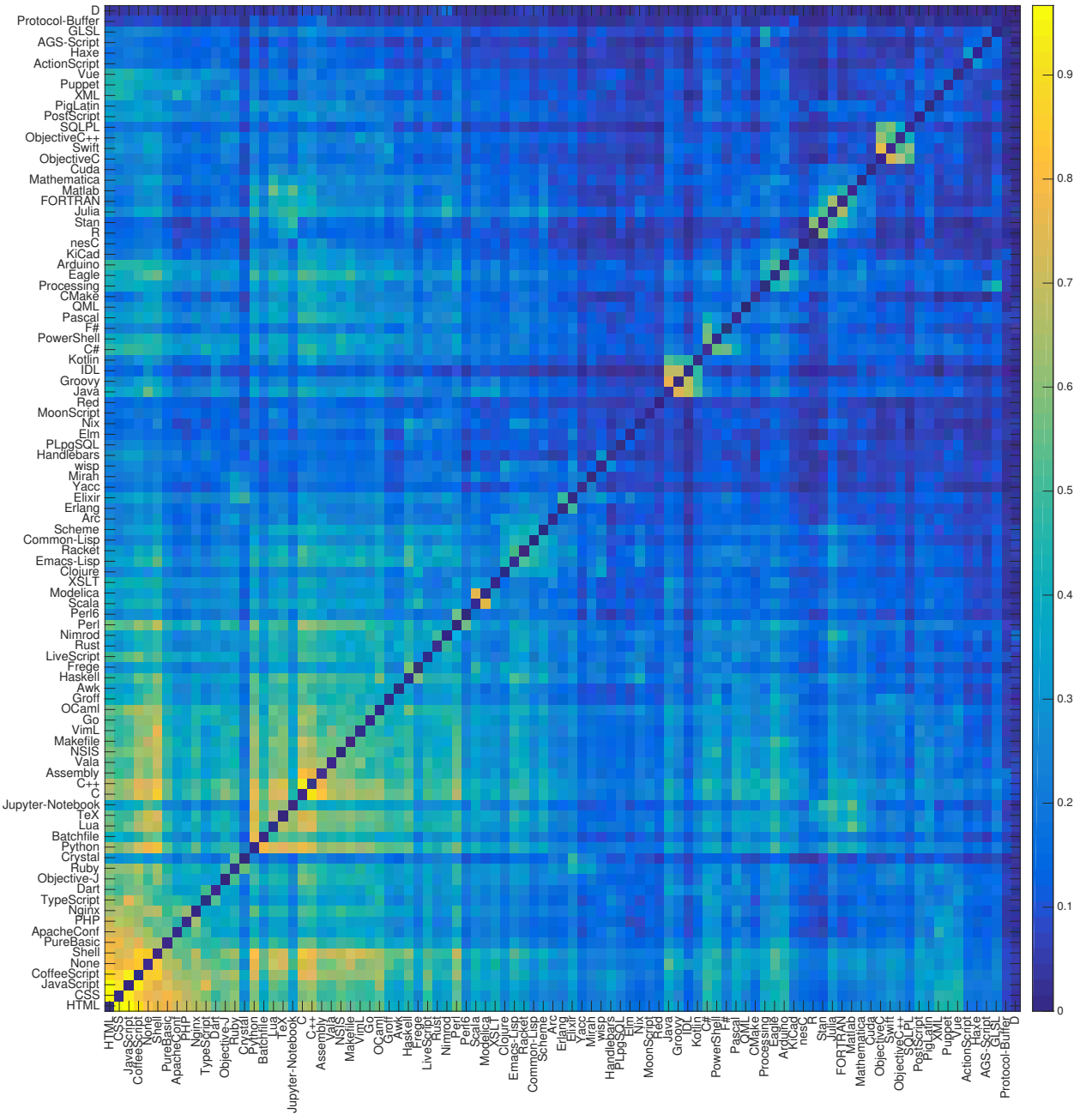


FIGURE 35: Heatmap of all programming languages using dot product-idf as repository distance and weighted as cluster distance

	U1	U2	U3	U4	U5	U6	U7
R1	1	0	1	0	0	0	1
R2	1	1	0	1	1	0	1
R3	0	1	0	1	0	1	1
R4	0	0	1	0	0	1	1

(a) Repository-user matrix

	U1	U2	U3	U4	U5	U6	U7
L1	1	1	1	1	1	0	1
L2	0	1	1	1	0	1	1

(b) Language-user matrix

TABLE 14: Repository-User matrix transform to Language-repository matrix

So We get the language-repository matrix and each language is represented as a vector. Then we can use the following mutual information to quantify the connection between programming language a and b:

$$MI(a,b) = \frac{\log \frac{P_{ab}}{P_a P_b}}{-\log P_{ab}} \tag{2}$$

where the probabilities P_a and P_{ab} is defined as follows. Suppose n_a is the number of users connect with programming languages a and n_{ab} is the common users between two programming languages a and b. N is the total number of users. Then $P_a = n_a/N$ and $P_{ab} = n_{ab}/N$. We add $-\log P_{ab}$ to normalize the value so that this value is in the range of -1 to 1. The positive value means there are more common users between these two programming languages than we expect, and a negative means there are less. As the large number of users, we removed those users whose degree is less than 3 from the network. So we calculate the mutual information of 95 programming languages on 592,259 users.

Then we plot the dendrogram tree of programming languages using mutual information as similarity measurement and the result is shown in Figure 36. This figure reveals some close pairs that are not so obvious, such as Julia&Fortran, Stan&R, Swift&ObjectiveC. Julia and Fortran are close because Julia supports the direct call of Fortran code. Stan is a statistic tool that is integrated with R. Swift is a successor of ObjectiveC. Elixir runs on Erlang virtual machine. Groovy interoperates with Java code and library, runs on Java Virtual Machine. Most valid Java programs are also valid for Groovy. There are also some obvious clusters of programming languages.

For instance, C is close to C++ and python . C++ is designed based on C and the implementation of Python is written by C.Besides, HTML, CSS, PHP, JavaScript is also close to each other and all of these programming languages are used to web developing.

The pairwise mutual information is plotted in Figure 37. Programming languages are sorted using hierarchical agglomerative clustering by average linkage. From this figure, we found the same result that web programming languages, functional programming languages and system programming languages separately compose a community.

- Functional programming: common-lisp, racket, scheme, frege, Wisp:
- OSX: ObjectiveC, Swift, Groff , ObjectiveC++, SQLPL
- System programming: C, C++, Go,Shell
- Web development programming: JavaScript, CSS, HTML, PHP, XML, CoffeeScript

3 Clustering using reduced dimensionality

Clustering high dimensional data is may not be accurate– high dimensions cause the distance large, and thereby the differences between pairs of entities be small. In subsections 1 and 2 , the dimension is separately in the order of 10^4 for repositories, and 10^6 for users. One remedy to the problem is to reduce the dimensions. Common approaches to dimensional reduction is PCA (principal component analysis), and more recently, t-SNE [22]. t-SNE (t-distributed stochastic neighbour embedding) is suitable for embedding high dimensional data to two or three dimensions. This is particularly good for us to visualize the relations between languages using scatter plot, so that we can plot similar languages by nearby points. In the following, we use t-SNE to reduce the dimensions.

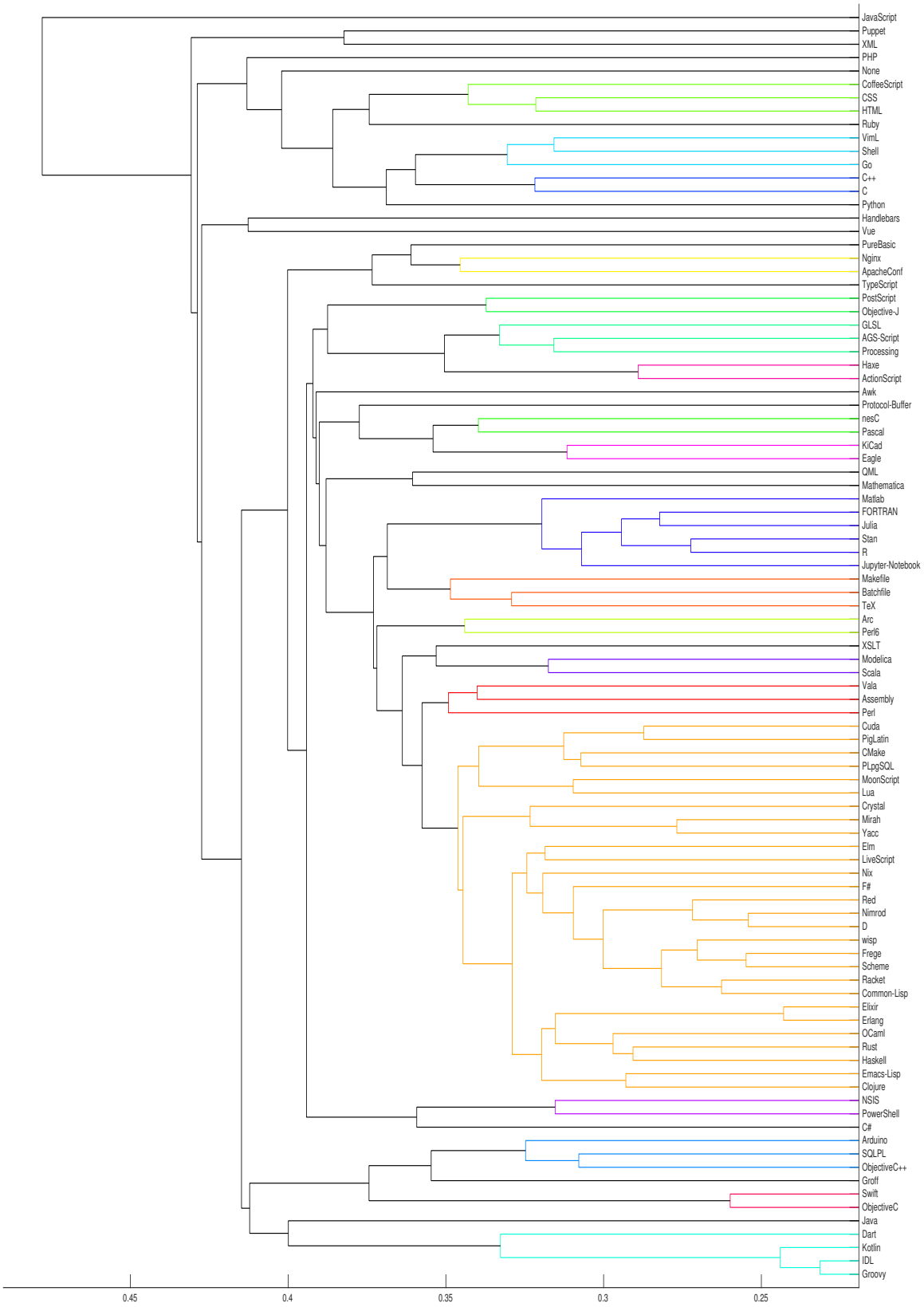


FIGURE 36: Clustering result using MI as similarity measurement

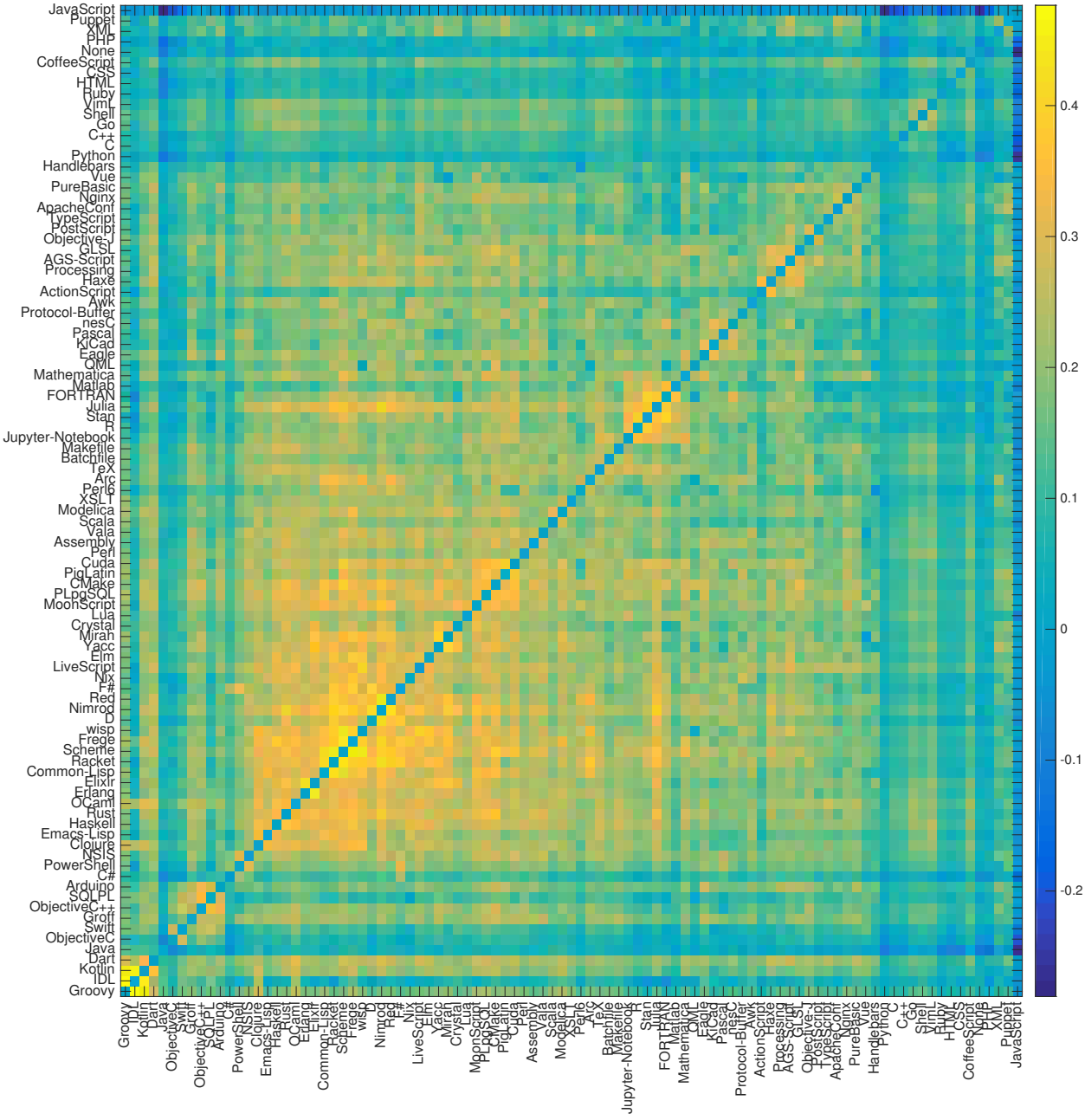


FIGURE 37: Mutual Information between Programming Languages

Given the Language-Repository $m \times n$ matrix M,

$$\begin{bmatrix} L_{1,1} & L_{1,2} & \dots & L_{1,n} \\ L_{2,1} & L_{2,2} & \dots & L_{2,n} \\ \dots & & & \\ L_{m,1} & L_{m,2} & \dots & L_{m,n} \end{bmatrix}$$

where m=94 and n=10,065. Each row is a vector representation of the language that is obtained in subsection 1. we run t-SNE to turn this matrix to a $m \times 2$ matrix as below, so that each language is represented by a two dimensional vector.

$$\begin{bmatrix} T_{1,1} & T_{1,2} \\ T_{2,1} & T_{2,2} \\ \dots & \\ T_{m,1} & T_{m,2} \end{bmatrix}$$

t-SNE is downloaded from https://github.com/lvdmaaten/bhtsne. The command to run the program is python bhtsne.py -i input file -o outpace -p 5 -d 2 -t 1 -v. The parameters include perplexity, dimension, and threads. One important parameter is perplexity and we tried perplexity = 5, which is the one commonly used.

Once the 2-dimensional vector representations are available for the languages, we run HAC using the Euclidean distance as defined by

$$d_{ij} = \sqrt{(T_{i,1} - T_{j,1})^2 + (T_{i,2} - T_{j,2})^2} \tag{3}$$

We run HAC using this distance function and use average as cluster distance. the resulting dendrogram and heatmap are shown in Figures 38 and 39. In addition to the dendrogram and the heatmap, we now can scatter-plot 2-dimensional embeddings of the languages in fig 40.

From these plots we can make the following observations.

- Overall clustering results are close to previous methods. We can see close pairs, such as C&C++, Objective-C&Swift, HTML& CSS

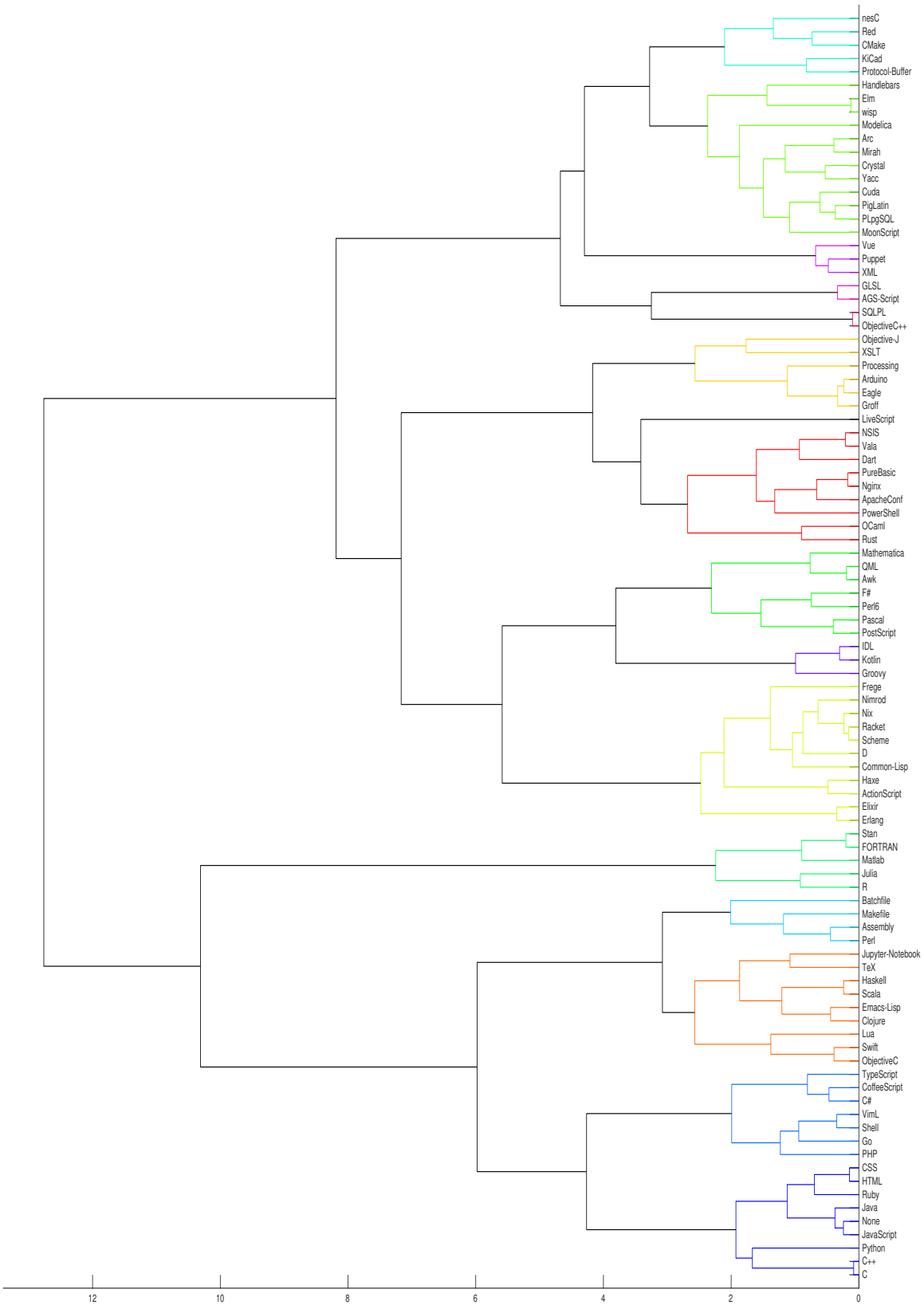


FIGURE 38: Languages clustered by Euclidian distance when dimensions are reduced to two using t-SNE

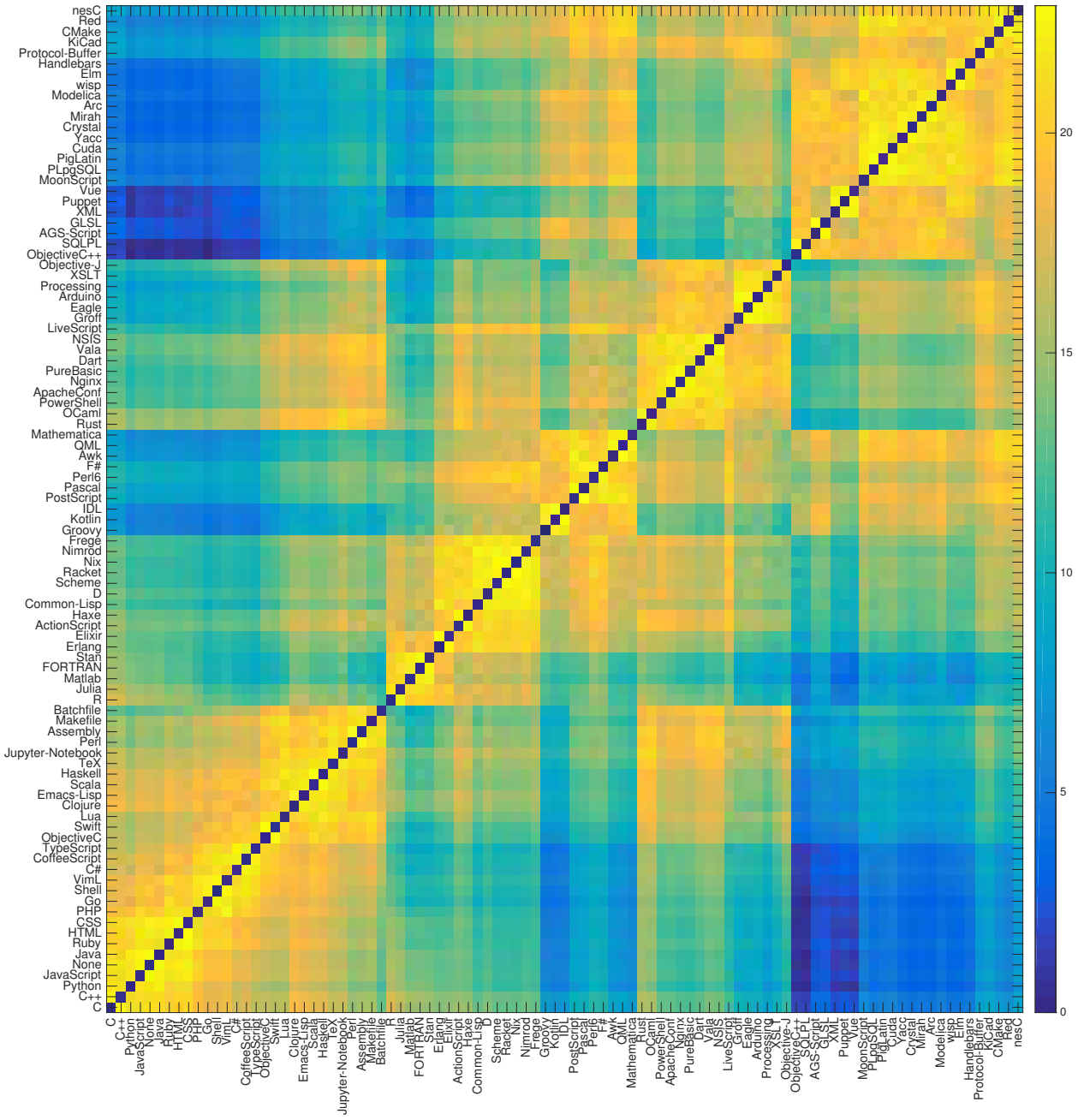


FIGURE 39: Heatmap of All Programming Languages based on language-language relation

- Figure 40 reveals more clustering information. We see, in both figures, som closely knit communities that are far away from any other languages, i.e., the group contains languages Matlab, R, Stan, Fortran, Julia.

4 Summary

We studied relations between languages based on distance between repositories and user interactions. Through the experiment, we found some interesting results. For example, no matter which method we used, C&C++, HTML&CSS, and Objective-C&Swift are always connected closely. An interesting phenomenon is FORTRAN, Matlab, Stan, and Julia are usually clustered in the same community. All of these programming languages are used for data analysis for statistical computing and numeric computation. For all the three different clustering method, we can see that most of the programming languages are clustered in the same communities but there is also some difference. For instance, only for clustering using repositories, Common-Lisp and Emacs-Lisp are close. While for the other two methods, these two programming languages are in different clusters.

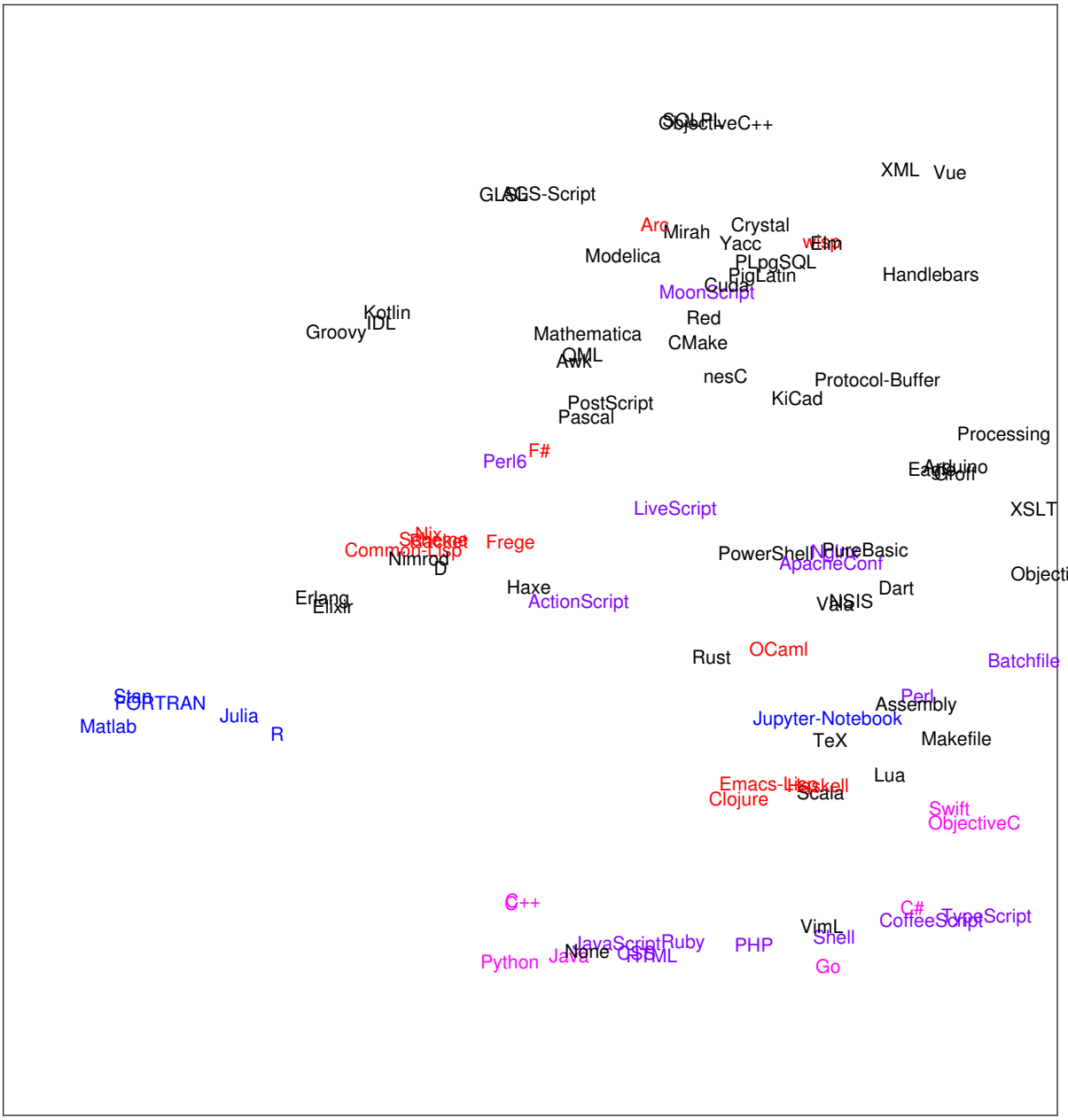


FIGURE 40: Scatter plot of two dimensional vectors of programming languages generated by t-SNE. Repository-repository proximity matrix is generated using dot product-idf weight

CHAPTER VIII

Conclusion and Future Work

In this thesis, we crawled around 12 million users from GitHub. Based on the user information, we collected over 1 million followers of users, around 10,000 popular repositories with at least 500 stars and the contributors and stargazers of these repositories. Then we discussed the heterogeneous-transformed homogeneous clustering method for the homogeneous network clustering. We selected three different categories weight, dot product, cosine similarity and Jaccard similarity to transform the heterogeneous network to a homogeneous one. Then we use modularity optimization algorithm cluster the homogeneous network and evaluated the result by F-measure and rand index. Through the experiments on the subdatasets of GitHub, we found that for the repository-stargazer network provides more information than the repository-contributor network. On the repository-stargazer network, the combination of dot product and inverse document frequency performs better than other weight schemes for both clustering algorithms. Next, we used dot product-idf as the weighting scheme to reveal the relationship between communities of the whole network. We separately detected 7 communities for greedy modularity algorithm and 4 communities for spectral modularity algorithm. From the result of both clustering algorithm, we found that there are some programming languages are usually belong to the same community, such web development programming languages(JavaScript, HTML, CSS and PHP), system programming languages(Python, C, C++ and Go) and programming languages for OS X and iOS system(Objective-C and Swift).

We also investigated the relation between programming languages on different clustering methods. One is based on the repository interaction. Transforming the

repository-repository matrix to language-repository matrix, each programming languages could be represented as a vector. Then we calculated the cosine distance between each vector and applied HAC (hierarchical agglomerative clustering) to get the hierarchical structure of programming languages. As the high dimensional of the language-repository matrix, the clustering result may not be accurate. Then we apply t-SNE to reduce the matrix to 2 dimensions and ran HAC using Euclidean distance as the input. Besides, due to the transformation may lose some information, we directly use the language-user relation to cluster the programming languages. This time, we utilized the mutual information as the distance function to evaluate the relation between programming languages. Form the experiment, it is difficult to decide which methods better performs the relation between programming languages, but we found some common phenomenon. Some pairs of programming languages are usually close, such as HTML&CSS, C&C++, and Matlab&FORTRAN. Overall, we find that languages fall into five communities, i.e., web and scripting languages (JavaScrip, HTML, CSS, etc.), system programming languages (C, C++, Python, etc.), OS X and IOS programming languages (Objective-C, Swift, Objective-C++, etc.), numerical and statistical languages (Matlab, FORTRAN, Julia and R), and functional programming (Lisp, Scheme, Racket, Haskell, etc.).

Future work will be addressed better understand the relation between programming languages and it can be summarized below:

- Data: the data is not large enough, especially the number of projects. When more projects are collected, we expect to have more programming languages. In addition to more comprehensive GitHub data, other data sources, such as GoogleCode, StatckOverflow, could be included.

- Language label: a project normally involves multiple languages. currently we use the top language to label the project. For each project, GitHub provides the lines of code for each language. The top language is the language that has the largest line of code. A more accurate analysis is needed for multi-labeled projects.

- Network embedding: we used t-SNE to reduce the dimensions. There are other dimensionality reduction techniques, in particular, network embedding techniques such as node2vec.

- Utilize user network: users are not isolated. they form a network by following links. this info could be utilized in network embedding.

- Compare with co-occurrence data: a project can use multiple languages. Thus, relations between languages can be quantified by their co-occurrences in projects. Existing analyses on co-occurrence data including the mining of association rules. This could be improved, and compared with the relation inferred from user interactions.

REFERENCES

[1] Github. `https://github.com/`.

[2] Github api. `https://developer.github.com/v3/`.

[3] Arthur, D. and Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics.

[4] Baeza-Yates, R., Ribeiro-Neto, B., et al. (1999). *Modern information retrieval*, volume 463. ACM press New York.

[5] Bastian, M., Heymann, S., Jacomy, M., et al. (2009). Gephi: an open source software for exploring and manipulating networks. *ICWSM*, 8:361–362.

[6] Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z., and Wagner, D. (2007). On finding graph clusterings with maximum modularity. *WG*, 7:121–132.

[7] Csardi, G. and Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal*, Complex Systems:1695.

[8] Delorey, D. P., Knutson, C. D., and Giraud-Carrier, C. (2007). Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects. In *Second International Workshop on Public Data about Software Development (WoPDaSD07)*.

[9] Fortunato, S. (2010). Community detection in graphs. *Physics reports*, 486(3):75–174.

[10] Getoor, L. and Diehl, C. P. (2005). Link mining: a survey. *ACM SIGKDD Explorations Newsletter*, 7(2):3–12.

[11] Girvan, M. and Newman, M. E. (2002). Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826.

[12] Gousios, G. and Spinellis, D. (2012). Ghtorrent: Github's data from a firehose. In *Mining software repositories (msr), 2012 9th ieee working conference on*, pages 12–21. IEEE.

[13] Guha, S., Rastogi, R., and Shim, K. (1998). Cure: an efficient clustering algorithm for large databases. In *ACM SIGMOD Record*, volume 27, pages 73–84. ACM.

[14] Han, J., Kamber, M., and Pei, J. (2011). *Data mining: concepts and techniques*. Elsevier.

[15] Huang, A. (2008). Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, pages 49–56.

[16] Huang, Y. and Gao, X. (2014). Clustering on heterogeneous networks. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4(3):213–233.

[17] Huang, Z. (1998). Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data mining and knowledge discovery*, 2(3):283–304.

[18] Karus, S. and Gall, H. (2011). A study of language usage evolution in open source software. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 13–22. ACM.

[19] Karypis, G., Han, E.-H., and Kumar, V. (1999). Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75.

[20] Kwak, H., Lee, C., Park, H., and Moon, S. (2010). What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM.

[21] Larsen, B. and Aone, C. (1999). Fast and effective text mining using linear-time document clustering. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–22. ACM.

[22] Maaten, L. v. d. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605.

[23] Newman, M. (2010). *Networks: an introduction*. Oxford university press.

[24] Newman, M. E. (2003). Mixing patterns in networks. *Physical Review E*, 67(2):026–126.

[25] Newman, M. E. (2004). Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066–133.

[26] Newman, M. E. (2006). Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582.

[27] Newman, M. E. and Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical review E*, 69(2):026–113.

[28] Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: bringing order to the web.

[29] Philip, S. Y., Han, J., and Faloutsos, C. (2010). *Link Mining: Models, Algorithms, and Applications*. Springer.

[30] Rand, W. M. (1971). Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850.

[31] Sanatinia, A. and Noubir, G. (2016). On github's programming languages. *arXiv preprint arXiv:1603.00431*.

[32] Sparck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21.

[33] Strang, G. (2011). Introduction to linear algebra.

[34] Strehl, A., Ghosh, J., and Mooney, R. (2000). Impact of similarity measures on web-page clustering. In *Workshop on Artificial Intelligence for Web Search (AAAI 2000)*, pages 58–64.

[35] Tang, J., Liu, J., Zhang, M., and Mei, Q. (2016). Visualizing large-scale and high-dimensional data. In *Proceedings of the 25th International Conference on World Wide Web*, pages 287–297. International World Wide Web Conferences Steering Committee.

[36] Ugander, J., Karrer, B., Backstrom, L., and Marlow, C. (2011). The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*.

[37] Vasilescu, B., Filkov, V., and Serebrenik, A. (2013). Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *Social Computing (SocialCom), 2013 International Conference on*, pages 188–195. IEEE.

[38] Wagner, S. and Wagner, D. (2007). *Comparing clusterings: an overview*. Universität Karlsruhe, Fakultät für Informatik Karlsruhe.

[39] Yin, X., Han, J., and Yu, P. S. (2006). Linkclus: efficient clustering via heterogeneous semantic links. In *Proceedings of the 32nd international conference on Very large data bases*, pages 427–438. VLDB Endowment.

[40] Zhang, T., Ramakrishnan, R., and Livny, M. (1996). Birch: an efficient data clustering method for very large databases. In *ACM Sigmod Record*, volume 25, pages 103–114. ACM.

VITA AUCTORIS

NAME:	Zhongpei Zhang
PLACE OF BIRTH:	Harbin, Heilongjiang province, China
YEAR OF BIRTH:	1989
EDUCATION:	Heilongjiang University, B.Eng., Computer Science and Technology, Harbin, China, 2012
	University of Windsor, M.Sc in Computer Science, Windsor, Ontario, 2016